

Interim
Final
Report

Volume 2

July 1990

Space Station Automation of Common Module Power Management and Distribution

(NASA-CR-184035) SPACE STATION AUTOMATION
OF COMMON MODULE POWER MANAGEMENT AND
DISTRIBUTION, VOLUME 2 Final Report (Martin
Marietta Corp.) 463 p

CSC 228

N91-12748

Unclas
0310575

63/20

MARTIN MARIETTA

MCR-89-516
Contract No. NAS8-36433

Interim
Final
Report

July 1990

SPACE STATION
AUTOMATION OF COMMON MODULE
POWER MANAGEMENT AND DISTRIBUTION

B. Ashworth
J. Riedesel
C. Myers
L. Jakstas
D. Smith

Martin Marietta Aerospace
Denver Astronautics Group
P.O. Box 179
Denver, Colorado 80201

This report was prepared by Martin Marietta Denver Astronautics Group for the National Aeronautics and Space Administration, George C. Marshall Space Flight Center (NASA/MSFC), in response to Contract, NAS8-36433, and is submitted as the Interim Final Report, as specified in the contract data requirements list. In particular, the work was performed for the Electrical Power Branch at NASA/MSFC.

Readers of this document are referred to NASA Contractor Report 4260 Space Station Automation of Common Module Power Management and Distribution and NASA Contractor Report 4273 Knowledge Management: An Abstraction of Knowledge Base and Database Management Systems.

[illegible]

Figure 1. Schematic representation of the experimental design. The subjects were divided into two groups: the control group (CG) and the experimental group (EG). The CG was divided into two subgroups: the control group (CG) and the control group (CG). The EG was divided into two subgroups: the experimental group (EG) and the experimental group (EG). The subjects were divided into two groups: the control group (CG) and the experimental group (EG). The CG was divided into two subgroups: the control group (CG) and the control group (CG). The EG was divided into two subgroups: the experimental group (EG) and the experimental group (EG).



Table of Contents

Interim
Final Report
Volume II

MCR-89-516
July 1990

1.0	INTRODUCTION	1 - 1
2.0	TESTBED 120 VOLT DC STAR BUS CONFIGURATION AND OPERATION	2 - 1
3.0	SSM/PMAD AUTOMATION SYTEM ARCHITECTURE	3 - 1
4.0	FRAMES RULES ENGLISH REPRESENTATION	4 - 1
5.0	THE SSM/PMAD USER INTERFACE	5 - 1
6.0	SSM/PMAD FUTURE DIRECTION	6 - 1

10/20/2020
10/20/2020
10/20/2020

10/20/2020
10/20/2020
10/20/2020

10/20/2020



Interim
Final Report
Volume II

MCR-89-516
July 1990

Appendixes

SSM/PMAD INTERFACE USER MANUAL VERSION 1.0	I-ii
SSM/PMAD LLP REFERENCE	II-1
SSM/PMAD TECHNICAL REFERENCE VERSION 1.0	III-ii
SSM/PMAD LLP VCLRs	IV-1
SSM/PMAD LLP/FRAMES ICD	V-1
SSM/PMAD LLP SIC ICD	VI-1



A/D	Analog to Digital Conversion
AC	Alternating Current
ACM/PMAD	Automation of Common Module Power Management and Distribution
AI	Artificial Intelligence
BLES	Baseline Load Enable Schedule
CAC	Communications Algorithmic Controller
CAS	Communication and Algorithmic Software
CLOS	Common Lisp Object System
DC	Direct Current
ECLSS	Environmental Control Life Support System
EMI	Electro-Magnetic Interference
EPLD	Erasable Programmable Logic Device
FELES	Front End Load Enable Scheduler
FRAMES	Fault Recovery and Management Expert System
GC	Generic Controller
H/W	Hardware
ICD	Interface Control Document
JSC	Johnson Space Center
KBMS	Knowledge Based Management System
KHz	KiloHertz
KNOMAD-SSM/PMAD	Knowledge Management and Design Environment applied to the SSM/PMAD Domain.
KVA	KiloVolt Amps
L-R	Inductive-Resistive
LC	Load Center
LES	Load Enable Scheduler
LISP	List Processing
LLF	Lowest Level Function
LLP	Lowest Level Processor
LPL	Load Priority List
LPLMS	Load Priority List Management System



MAESTRO	Master of Automated Expert Scheduling Through Resource Orchestration
MMAG	Martin Marietta Astronautics Group
MSFC	Marshall Space Flight Center
NASA	National Aeronautics and Space Administration
NDA	Node Distribution Assembly
OMS	Operational Management System
PCL	Portable Common Loops
PCU	Power Control Unit
PDCU	Power Distribution Control Unit
PLES	Preliminary Load Enable Schedule
PPDA	Primary Power Distribution Assembly
RBI	Remote Bus Isolator
RCCB	Remote Control Circuit Breaker
RMS	Root Mean Square
RPC	Remote Power Controller
SADP	Systems Autonomy Demonstration Program
SDA	Subsystem Distributor Assembly
SI	Symbolics Interface
SIC	Switchgear Interface Controller
SPST	Single Pole Single Throw
SSM	Space Station Module
SSM/PMAD	Space Station Module Power Management and Distribution
SSS	Supervisor Subsystem Simulator
S/W	Software
TCP/IP	Transmission Control Protocol / Internet Protocol
UI	User Interface
VCLR	Visual Control Logic Representation



1.0 INTRODUCTION

Space Station Module Power Management and Distribution System (SSM/PMAD)

1.1 Scope

This report is intended to describe the new SSM/PMAD testbed automation system. The system, with automation hardware and software elements, was most recently delivered to NASA, George C. Marshall Space Flight Center in June, 1990. All product names are trademarks of their manufacturers.

1.2 Coverage and Changes

Since the SSM/PMAD Interim Final Report was delivered in February of 1989 several significant changes have occurred. These are:

- 120 Volt dc Star Bus Configuration
- New general purpose workstation hardware (Solbourne 5/501)
- New multi-tasking workstation operating system (UNIX)
- New LLP hardware (Intel 80386 based processors)
- New Communications protocol and hardware (Ethernet TCP/IP)
- Parallel knowledge base management system (KBMS)
- Integrated database/knowledge management system (KNOMAD-SSM/PMAD, – a specialized KBMS)
- Rule organization for the FRAMES expert system
- Handling of multiple faults
- Cooperative interaction capabilities using the KNOMAD-SSM/PMAD
- Significant improvement in LLP transactions and fault management
- Significant improvement in the User Interface

Each of the topics will be reported on in this IFR Update which is intended to accompany the June 1990 delivery. The testbed changes and enhancements reported on include the October 1989 and June 1990 deliveries. Readers of this document are referred to the list of acronyms and abbreviations at the front of the report.

1.3 Overview

Following the December 1988 delivery of the SSM/PMAD automation system, work immediately began to update and improve the automation software. NASA's directive for 120 Vdc on the Space Station Freedom was in effect, and several weaknesses in the delivered FRAMES software had surfaced. The fault management knowledge needed reorganization, and management of the knowledge entities in the testbed was not present. The user interface needed to be made seamless with respect to the computer operating systems and languages being used. Some of the needed enhancements and

developments were delivered in October 1989 and June 1990; others are to be delivered in December 1990.

The original SSM/PMAD automation system contained several key weaknesses. These were:

- Non multi-tasking operating system interactions leading to poor performance.
- Special purpose AI system hardware which provided weak communications systems and multi-tasking support.
- LISP function level knowledge representation which impeded system performance and promulgated rigidity.

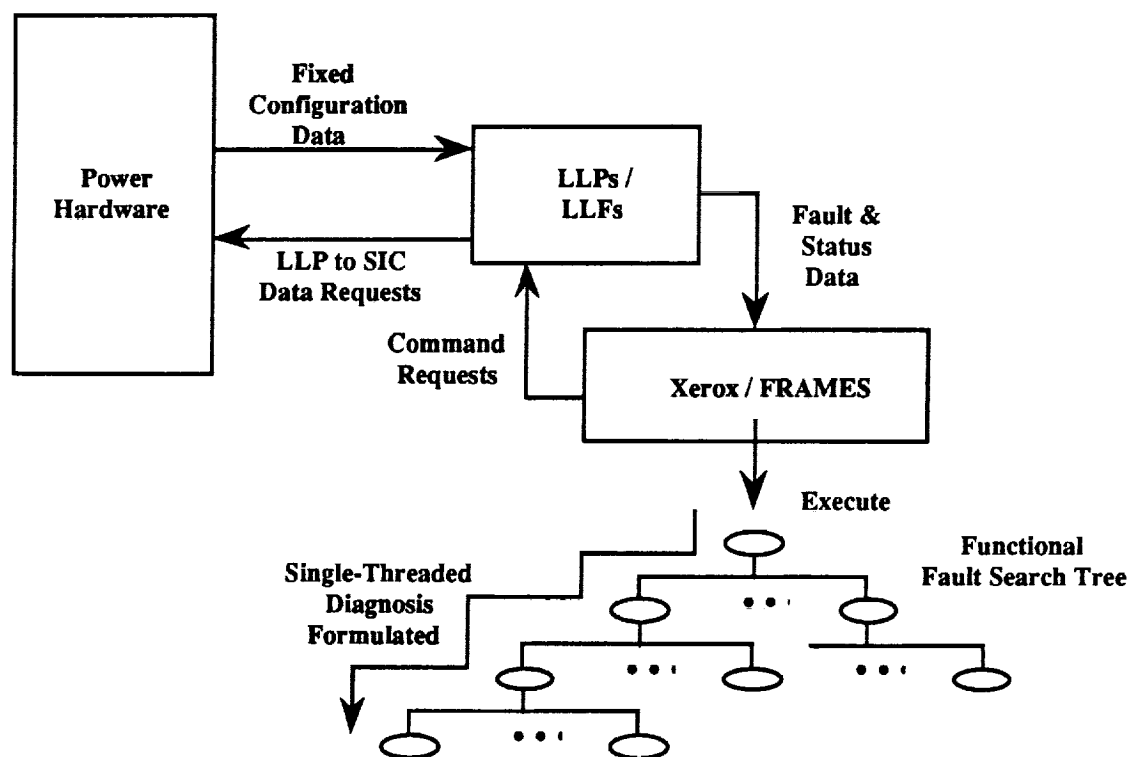


Figure 1.1-1. Past SSM/PMAD Fault Isolation Methodology

The solution to the system weaknesses was to introduce new general purpose hardware to replace the Xerox 1186 AI workstation, the VME/10 CAC workstation, the MVME 107 LLPs, and to reorganize and enhance the software and knowledge representations within the system.

Figure 1.1-1 depicts the fault isolation methodology used in the past SSM/PMAD system. A functional path through a decision tree was executed until the fault was identified. This presented many obstacles when multiple faults or faults in a non-well-behaved system occurred. It also made updating of knowledge processing activities very difficult as the introduction of new functions affected all of the processing activities underneath. Essentially, the LISP read-eval-print control loop also functioned as the rule-interpreter.

Figure 1.1.2 shows the new knowledge control mechanism which leads to a new fault isolation methodology. Rules and objects are treated as knowledge entities by the KNOMAD-SSM/PMAD knowledge manager. The rules and objects exist within the knowledge bases, which can be executed as parallel agents. This provides a strong environment for cooperating expert systems.

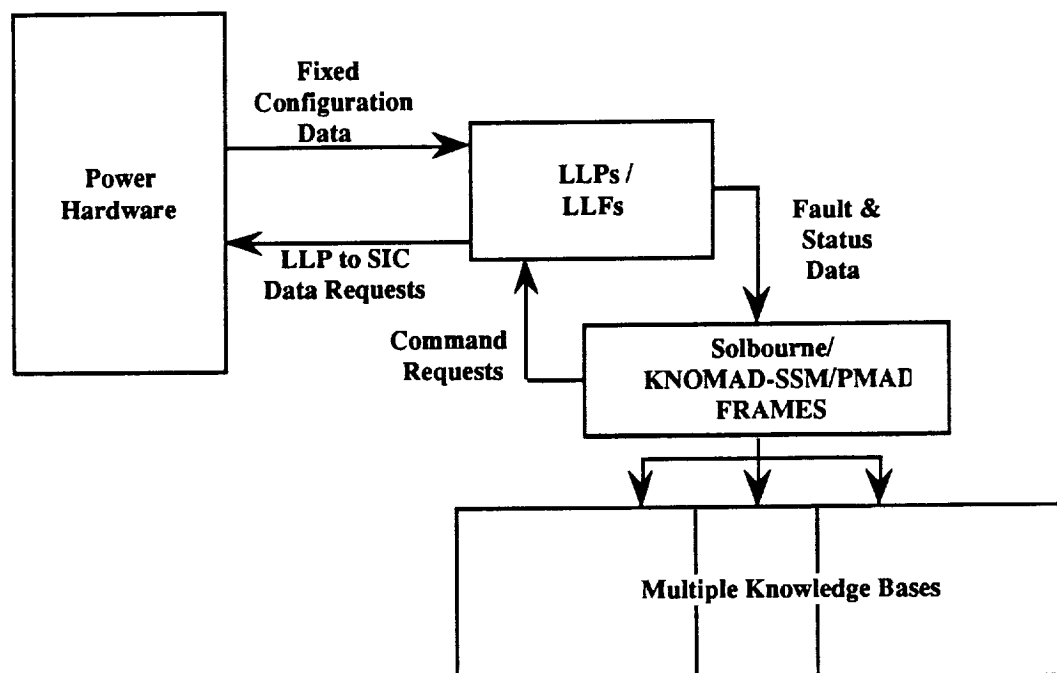


Figure 1.1-2 Present SSM/PMAD Fault Isolation Methodology

The management of knowledge objects and data provided by the KNOMAD system is a key element to the increase in performance (from 10 to 200 times) of the overall SSM/PMAD system. Also adding to the increase were the introduction of ethernet as the communications link between the Solbourne 5/501 workstation and the LLPs; and the use

INTRODUCTION

of 80386 based processors for the LLPs. Perhaps the single biggest change was the use of the Solbourne 5/501 workstation employing the UNIX multi-tasking operating system. Allegro Common LISP is used. When the present automation system is compared to the Xerox 1186 and its native InterLISP operating environment from the past SSM/PMAD system, gains are made in development time, time spent on test and debug, overall system capabilities (such as graphical interface), and execution performance. KNOMAD-SSM/PMAD provides much greater flexibility in updating knowledge bases.

1.4 Organization

Sections and appendixes in this report will describe the 120 Vdc STAR bus topology, the new system architecture, the new knowledge management mechanism, the new LLP capabilities, the new user interface, and future directions. All significant technical detail will be supplied in the appendixes.

2.0 TESTBED 120 VOLT DC STAR BUS CONFIGURATION AND OPERATION

2.1 Configuration

The first version of the SSM/PMAD testbed was based on 208 Volts, 20 KHz ac configured in a ring bus topology. In December, 1988, NASA specified a 120 Volt dc source for the Space Station Freedom. Also specified was a change to a star power bus topology. Figure 2.1-1 depicts the resulting SSM/PMAD testbed organization following these NASA specified changes.

Rules within the FRAMES knowledge base were changed as was the lower level FRAMES at the LLPs. Also changed were the system models at the Symbolics 3620D and the Solbourne 5/501 Workstation.¹

The resulting system-wide architecture is favorable to the distributed Lower Level Processors. At the Solbourne Workstation the communications handling segment manages the messages to and from the LLPs. KNOMAD-SSM/PMAD handles distribution of knowledge and database updates. FRAMES isolates and handles fault situations and distributes schedule segments to appropriate LLPs.

When the source changed from ac to dc there was no longer need to integrate at the switch hardware level to calculate the power factor. Therefore, the power factor (a nominal value of 1) and the system power is appropriately calculated at the LLP as $P=VI$.

2.2 Operation

The operational flow of the SSM/PMAD testbed consists of functional interactions between the various computational and control elements which result in hardware commands. System flow diagrams show these activities in Figures 2.2-1 through 2.2-7 and 2.2-9 through 2.2-11.

Intersystem communication between the SSM/PMAD testbed and other testbeds is crucial if the SSM/PMAD testbed is to serve as a realistic prototype for the Space Station Freedom. A candidate power profiling configuration for communications items is depicted in Figure 2.2-8. The dynamic items to be handled are the priorities of the loads to be powered and the minimum and maximum power which is to be supplied to those loads according to the associated priorities.

For the operation of the testbed to be meaningful during management of different faults, it is necessary for there to be a single process definition for how fault information is managed, and how fault isolation information is gathered. Figure 2.2-12 shows the fault isolation process used when FRAMES interrogates the LLPs in the SSM/PMAD testbed.

¹ The Xerox 1186 AI workstation has been replaced by a Solbourne 5/501 general purpose workstation

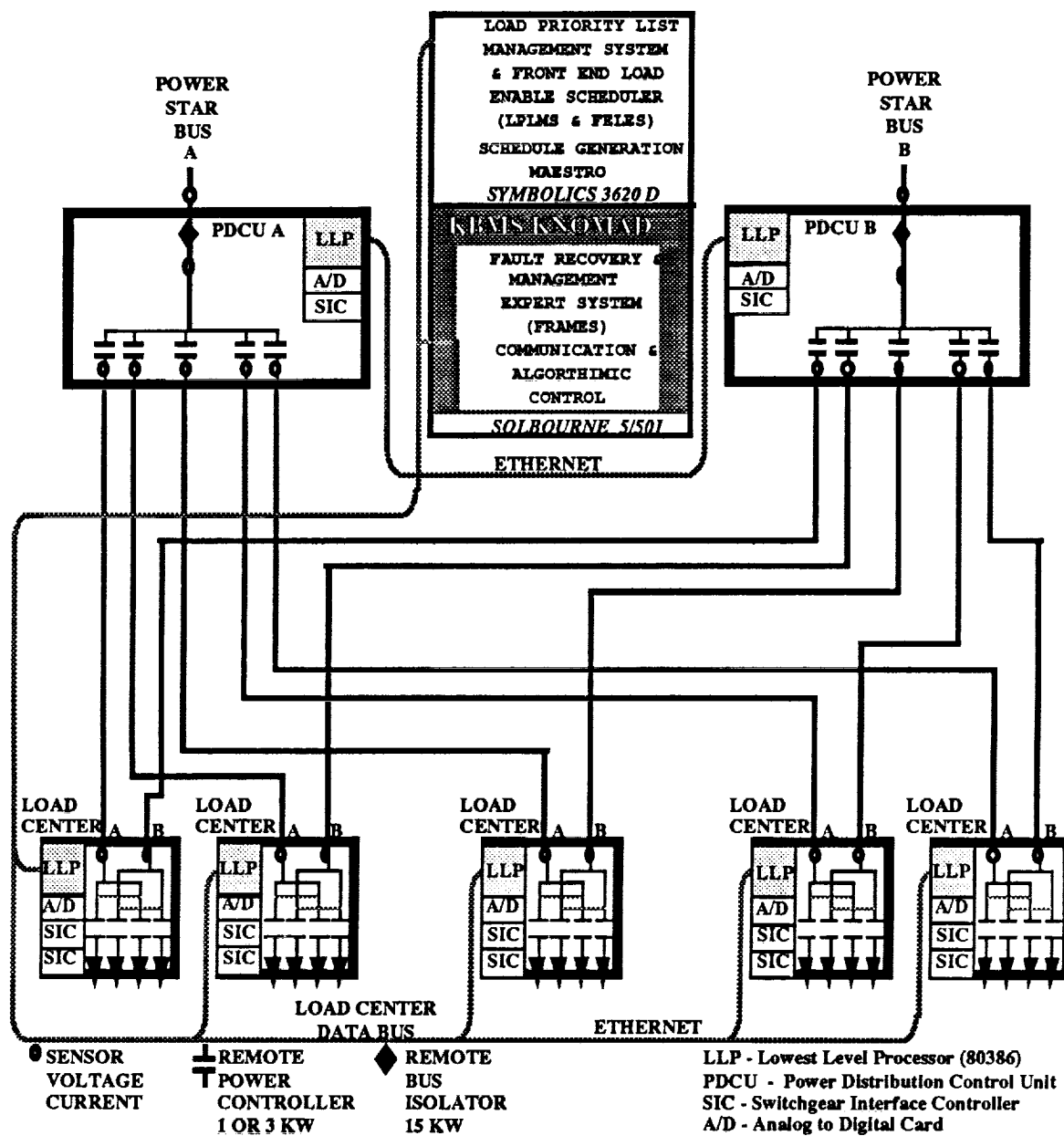


Figure 2.1-1 The SSM/PMAD Block Diagram

Big Picture 1

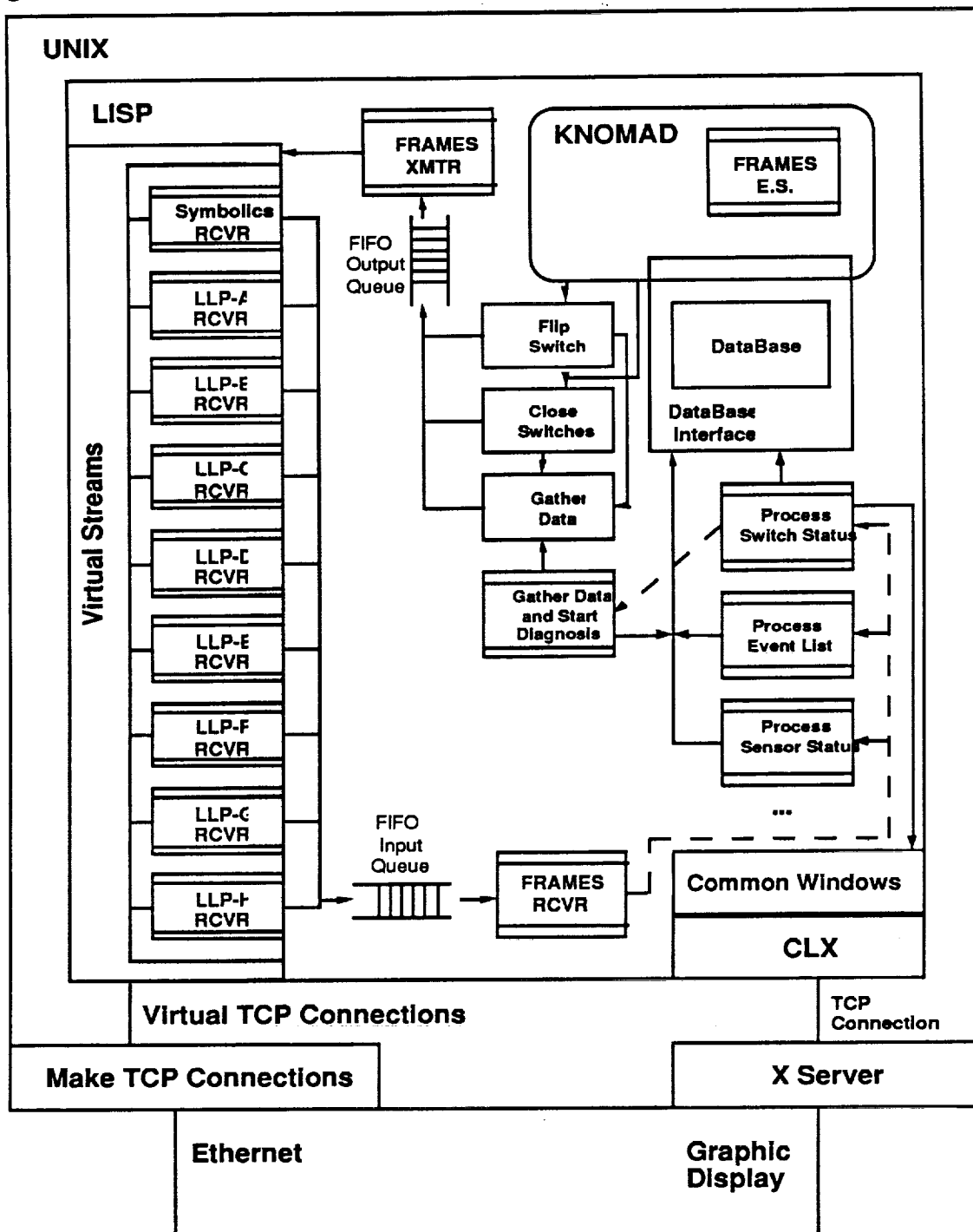
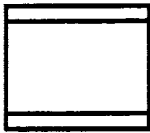


Figure 2.2-1 SSM/PMAD FRAMES Overall Operational Flow

Flowchart and Big Picture Explanations



Process. Used to represent a process running in parallel with other processes. Consists of an algorithm.



Function. Used to represent a basic function. Executes in a process.



Dashed arrow represents a "fork". A fork is where one process starts another process and then continues on while the started process runs in parallel with the original process.

variable

Global variable (even over processes)

The Big Picture.

The big picture represents a lot of what is happening on the Solbourne. The functions associated with updating the user interface are not represented. Most of the communications processes are represented.

Basically, what is happening is that there are nine processes listening for messages from the other processors. As a message is received it is queued up on the input queue and the FRAMES receiver is signalled. The FRAMES receiver then determines what kind of message it is and starts off a process for that message. Messages usually require updating the database as well as the user interface. If there is a fault signalled, a gather-data-and-start-diagnosis process is started. This process uses the gather-data routine to actually wait for and query for data. These routines are described in the accompanying flowcharts.

When a query is made or command performed, the message is put on the output queue and the transmitter is notified. Additionally, as a part of fault diagnosis, commands to flip or close switches may be executed – which involves a call to the gather-data routine to see what happens.

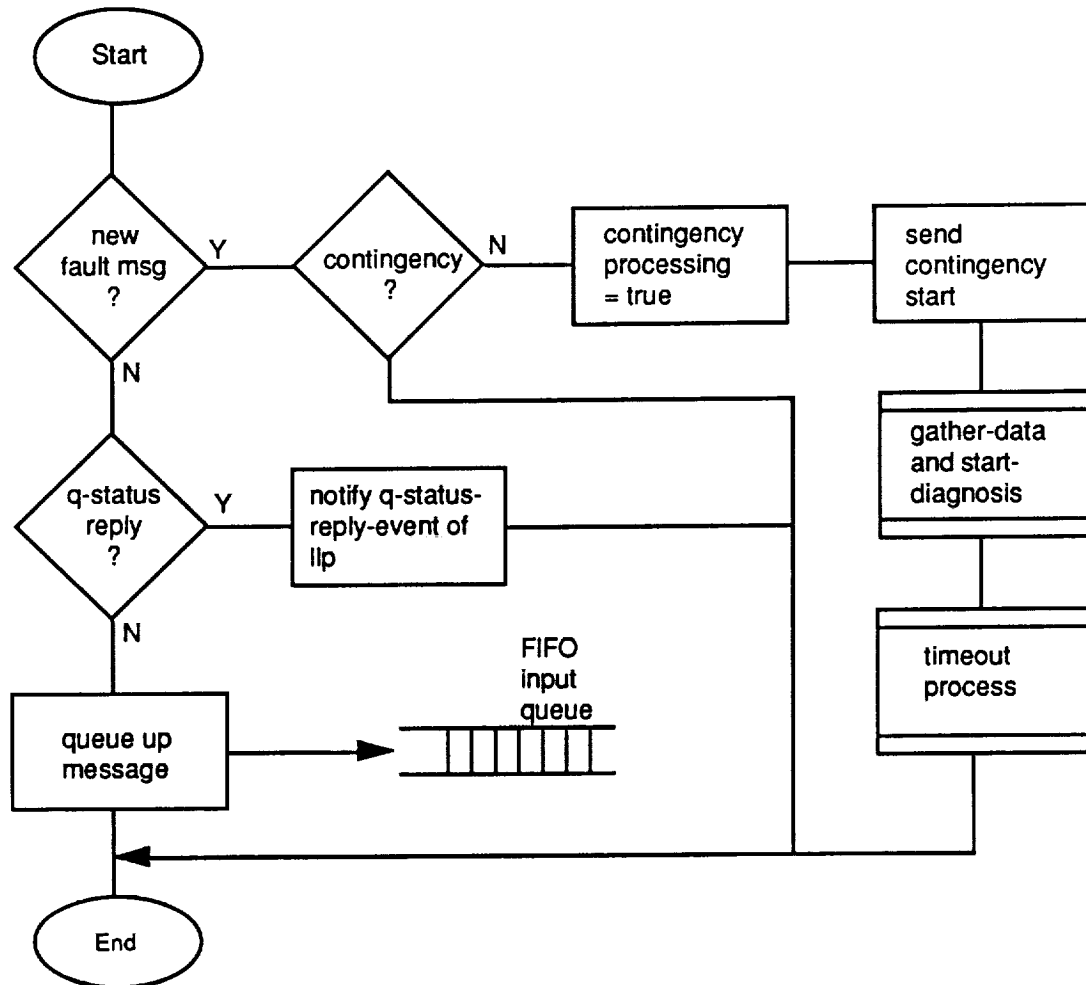
The Architecture.

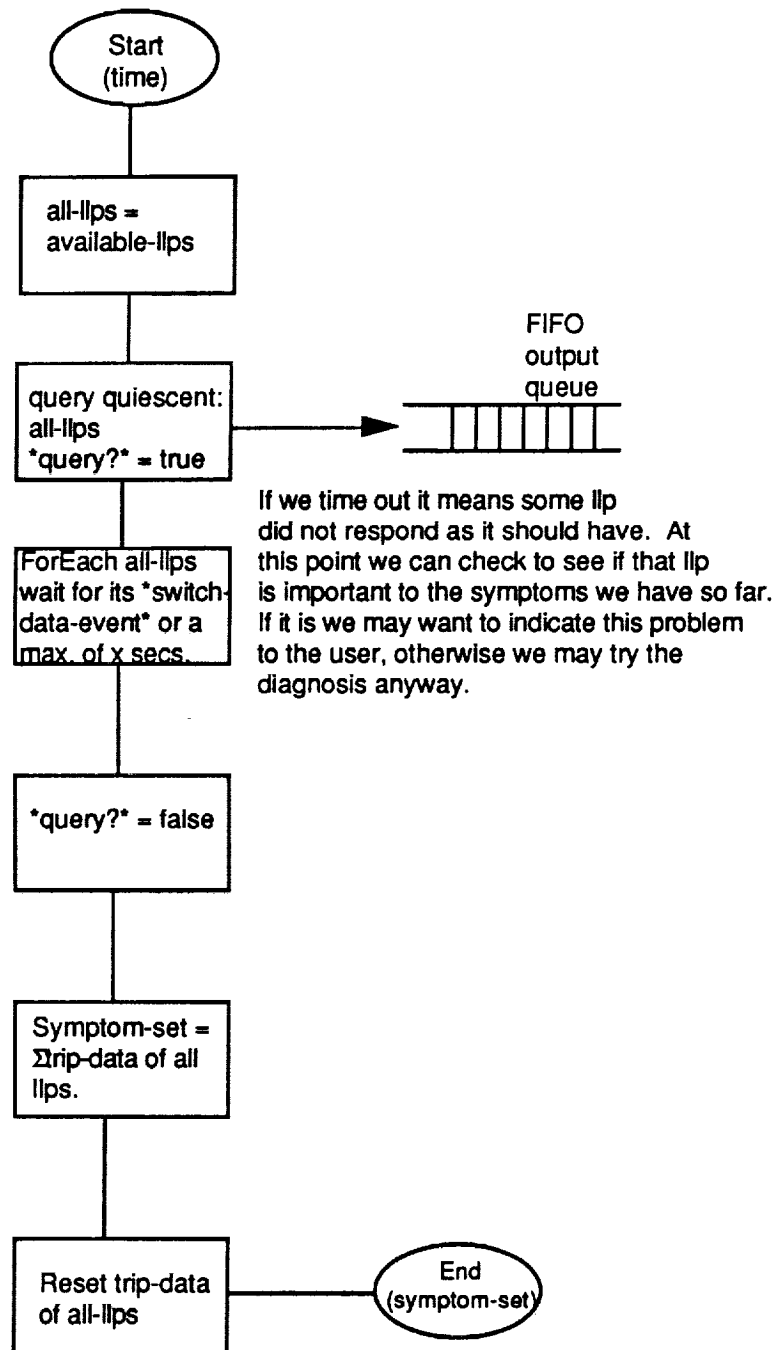
An important implementational consideration is the architecture of what processes are running and what environment are they running in. As the Big Picture shows, most of the processes for the user-interface and data collection are performed in LISP. To actually talk to the outside world requires an interface to UNIX (using C functions) for ethernet communications and for executing user interface functions.

Let's look at this in a bit more detail. UNIX manages multiple tasks by scheduling amounts of CPU time between the processes it is managing. In this case it is the X server, communications functions, and LISP. LISP also allows multiple tasks to be run concurrently. LISP also uses a time-slicing mechanism as does UNIX. Thus we can see that as more processes are run under LISP, the amount of time each process gets of the CPU diminishes as a function of LISP AND UNIX time-slicing.

An important option is to move the data collection and user interface processes to the UNIX environment in C functions. This option then allows a single level of time-slicing for the data collection and user-interface processes, in effect, allowing greater throughput in these tasks. I, personally, believe that the Solbourne can handle the tasks if implemented in this manner as opposed to everything running under LISP. KNOMAD, however needs to stay in the LISP environment.

Another comment about the user-interface is that if the user interface functions were moved to the UNIX environment as C functions, the extra two layers of abstraction (Common Windows and CLX) would be eliminated. This has the potential of significantly speeding up the user interface (and therefore also allowing more functions to be easily supported at the user interface).

LLP Receiver**Figure 2.2-2** *FRAMES handling of LLP Input*

Gather Data**Figure 2.2-3** *FRAMES Gather Data Process Flow*

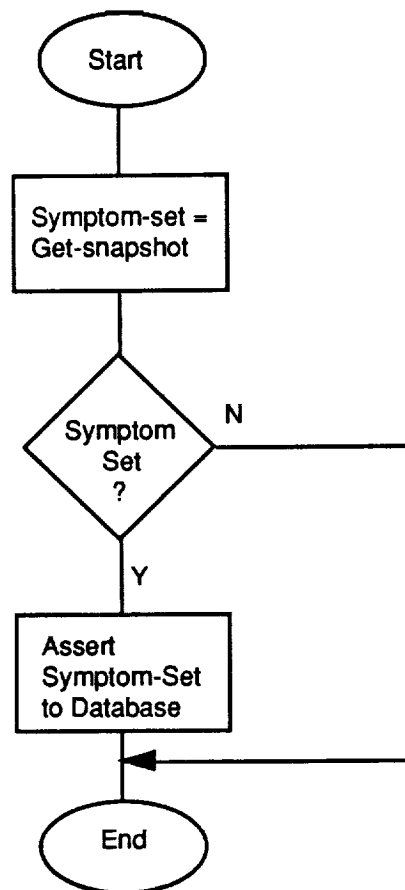
Gather Data and Start Diagnosis

Figure 2.2-4 *FRAMES Isolation of Symptom Sets*

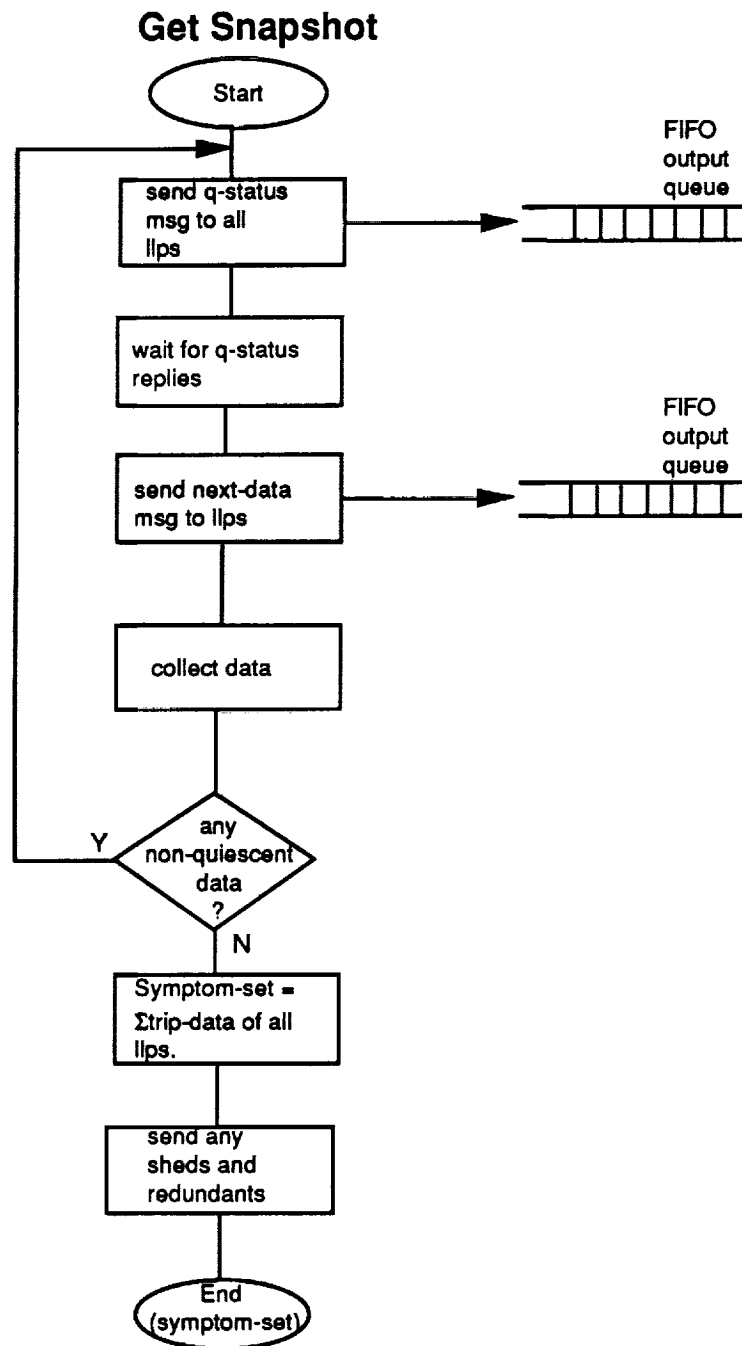
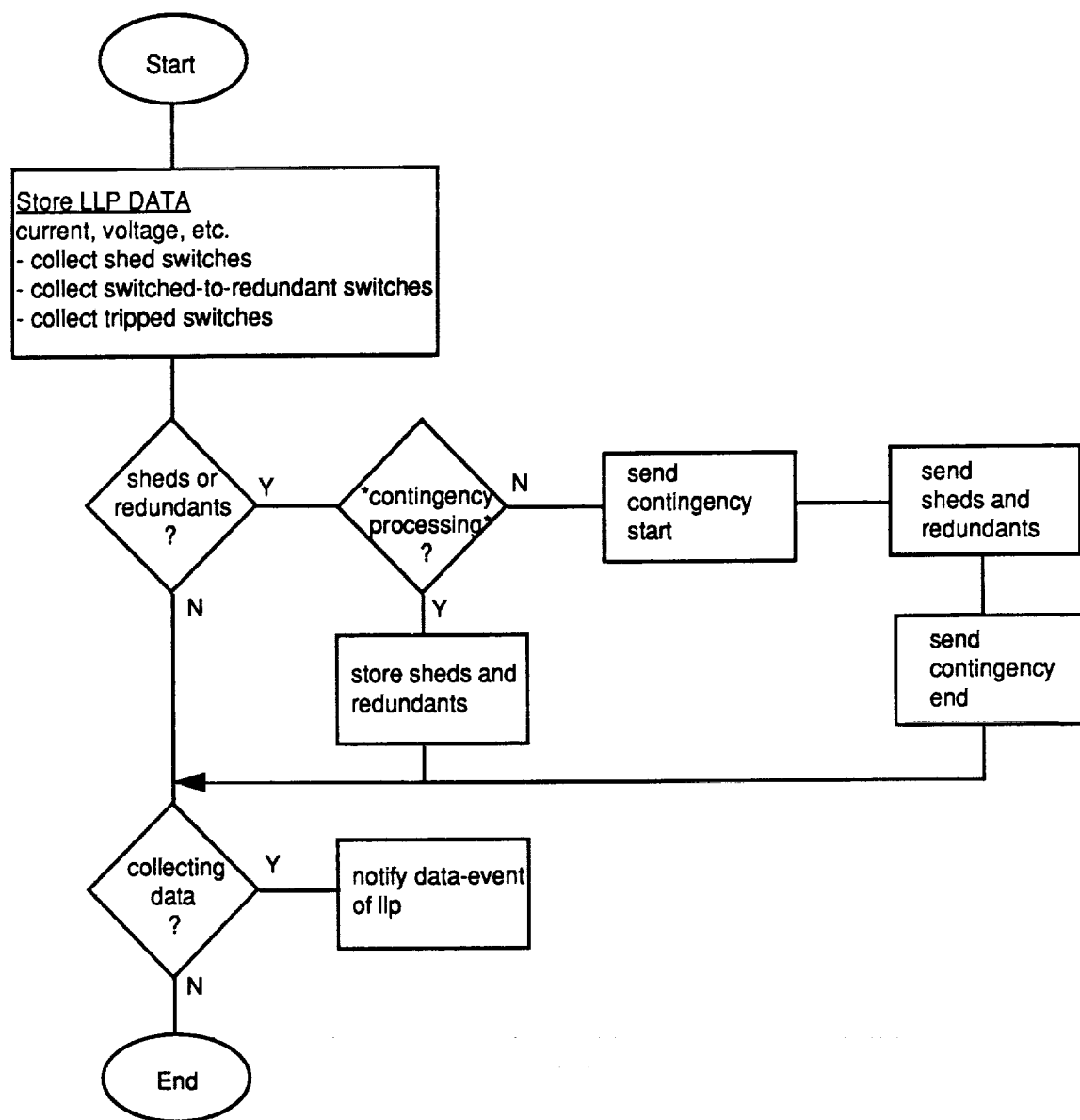
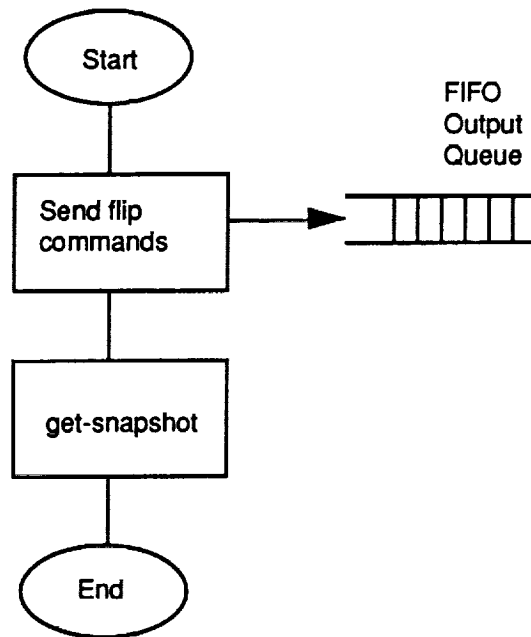
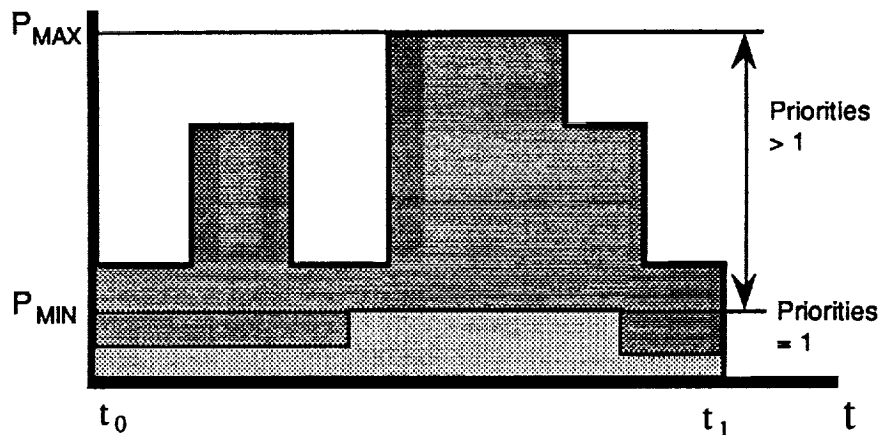


Figure 2.2-6 *Snapshot Data from LLPs to FRAMES*

Process Switch Status**Figure 2.2-6** *FRAMES Processing of Switch Status Information*

Flip Switches**Figure 2.2-7 FRAMES Commanding Switches**

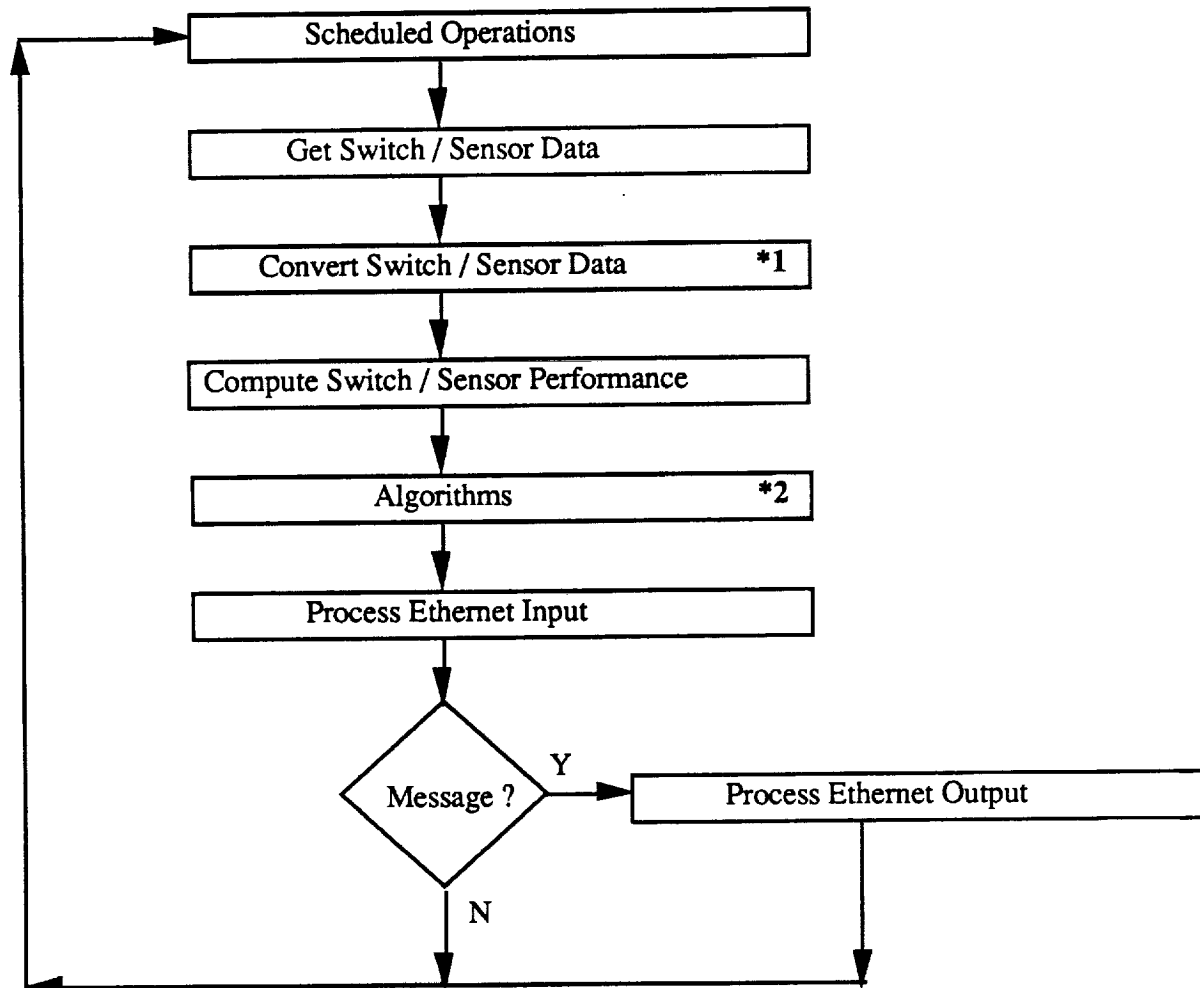
Load Profiles of 1 minute granularity for 2 buses

Priority Profiles as Updated for 2 buses *

 P_{MIN} (Minimum power that must be supplied) P_{MAX} (Maximum power that can be supplied)

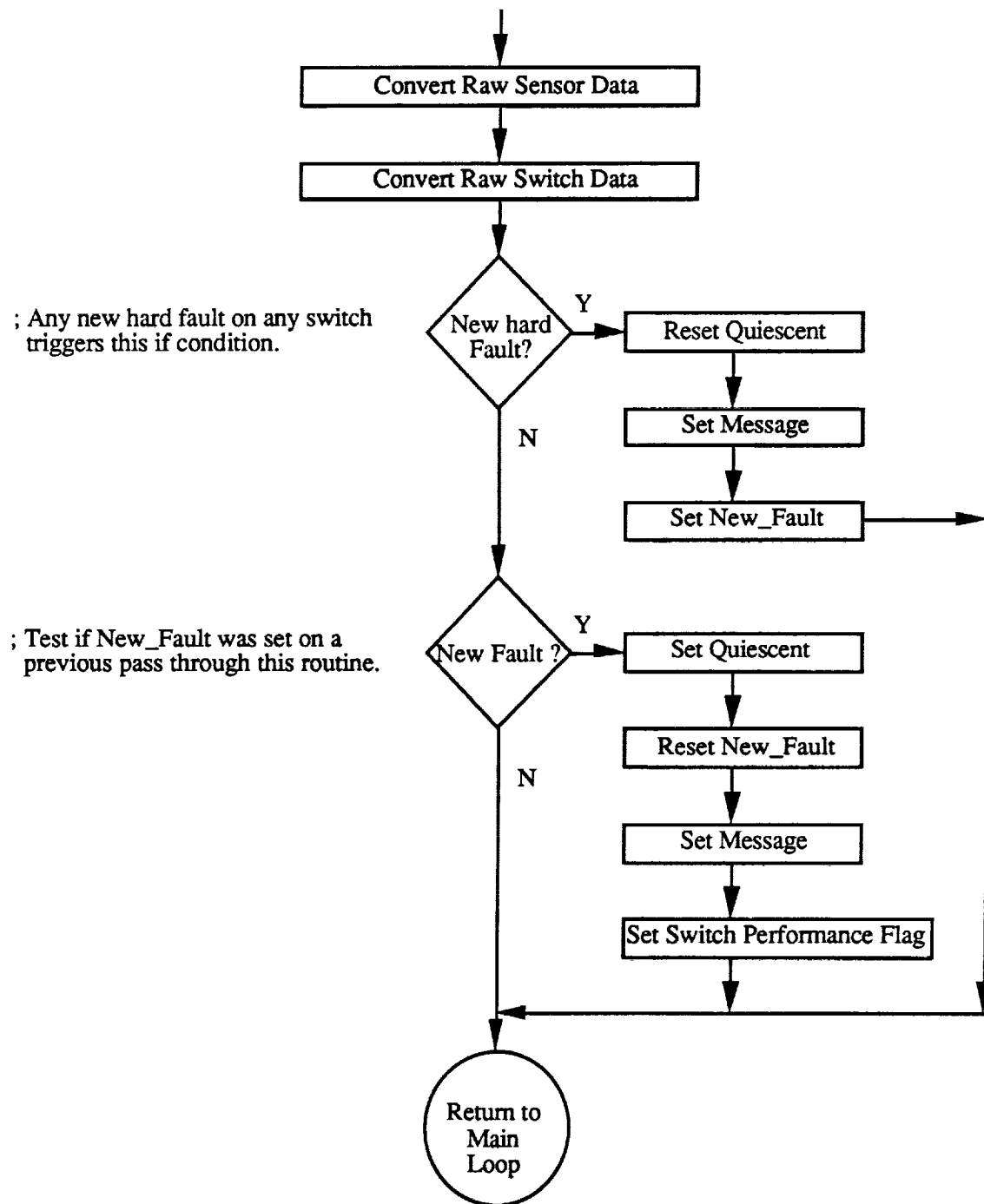
* Priority profiles are provided on a per priority basis.

Figure 2.2-8 The SSM/PMAD Power Profiling Configuration



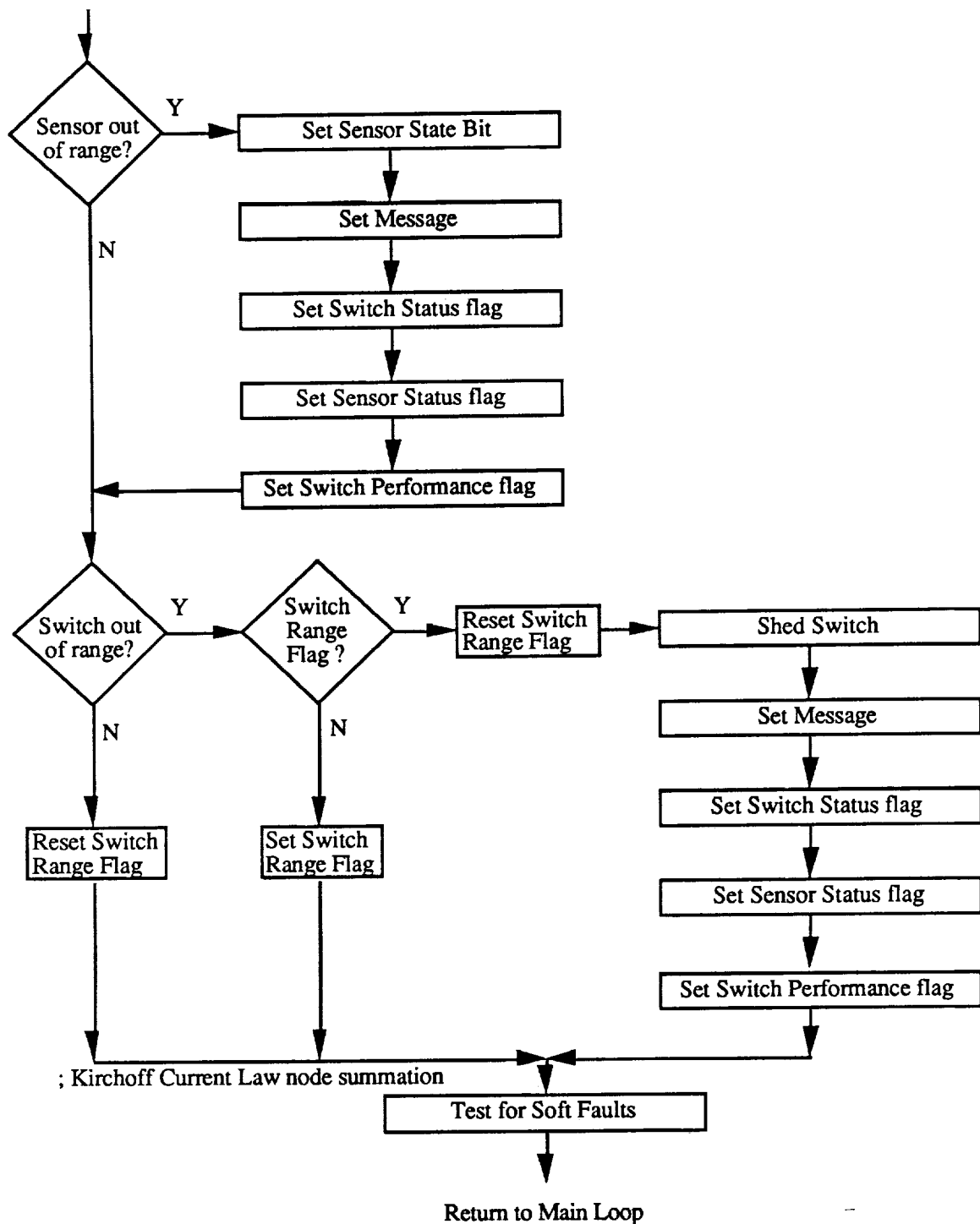
Lowest Level Processor Main Loop after Initialization

Figure 2.2-9 *LLP Main Loop Processing*



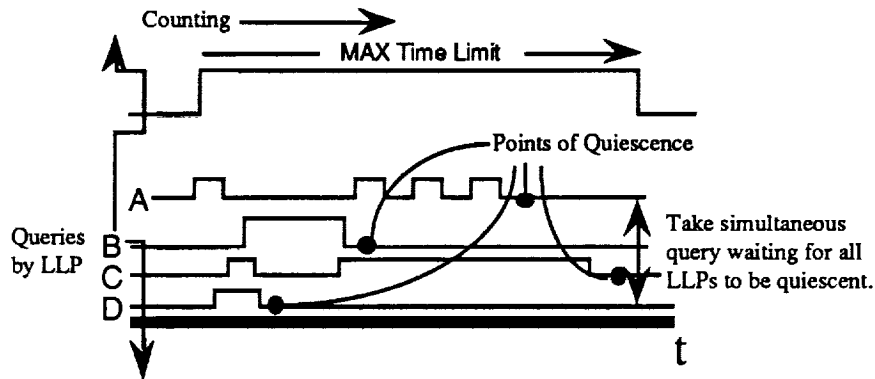
**1 : Convert Switch / Sensor Data*

Figure 2.2-10 *LLP Process Loop *1*



*2 : Algorithms

Figure 2.2-11 LLP Process Loop #2

**DEFINITIONS:**

Quiescent Data - fault status of faulted switches are identical after two consecutive reads.

End of Fault - data is quiescent within an LLP.

End of Event - data is quiescent in all LLPs simultaneously.

Symptom Set - data gathered from LLPs after an end of event.

QUANTIFIABLE EXPRESSIONS:

E \equiv Event as reported by the LLPs.

Q \equiv Quiescent State as detected at the LLPs.

$= 1$ if End of Fault is true or no fault exists;

0 otherwise.

$C \equiv$ fault condition $= Q_1 \cdot Q_2 \cdot Q_3 \cdot \dots \cdot Q_n$

$Q_m \Leftrightarrow$ last buffered value of Q_{LLP} ;

QUIESCENCE CONDITIONS:

$QA = 0$ if fault scan $N \neq$ fault scan $N-1$;

else, If $E = \text{true}$ then $QA \Leftrightarrow 1$.

SYMPTOM SET DETERMINATION:

If *query response* for all $Q=1$,

then query for *initial symptom set and end of events*;

else, $C = 0$.

Figure 2.2-12 **Fault Isolation Process and Definitions**

3.0 SSM/PMAD AUTOMATION SYSTEM ARCHITECTURE

The automation system architecture has undergone a radical change due to the introduction of a new hardware workstation, new LLP hardware, and new systems software. Figure 3-1 shows the old automation system information flow with respect to systems software. As can be seen, any operation to or from the switchgear hardware required passage through several different operating environments and communications systems.

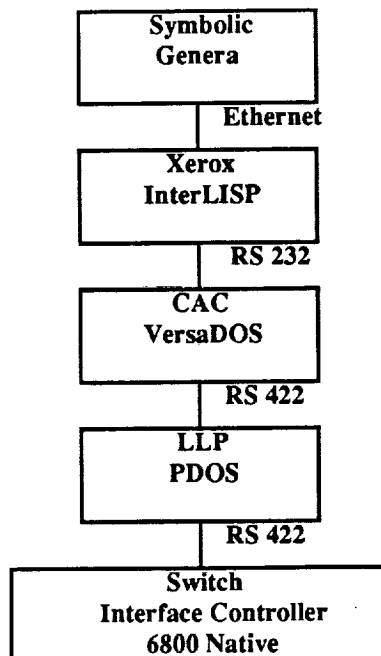


Figure 3-1 Previous SSM/PMAD Systems Level Architecture

In order to improve the performance of the automation portion of the testbed several things had to be accomplished. First, the number of hardware interactions to carry out switchgear operations had to be decreased. Second, one-way bottlenecks in communications needed to be eliminated. Last, a more streamlined way of handling automation data functions were to be introduced. The automation system which was delivered to NASA/MSFC in June accomplished these three goals.

For the new system the number of operating components has been reduced. Likewise, the communications overhead has been reduced and the protocol has been made more homogeneous than before. The new architecture for the informational flow of the testbed is shown in Figure 3-2. Multitasking advantages have been introduced by using the UNIX operating system at the Solbourne workstation. The LLPs function with dedicated processes which do not need to exercise multitasking within the system. Therefore, they function as resident processes under MS/DOS using operating system features only as interfaces to the Ethernet and RS422 drivers. The LLPs are rack-mounted with no keyboards or monitors.

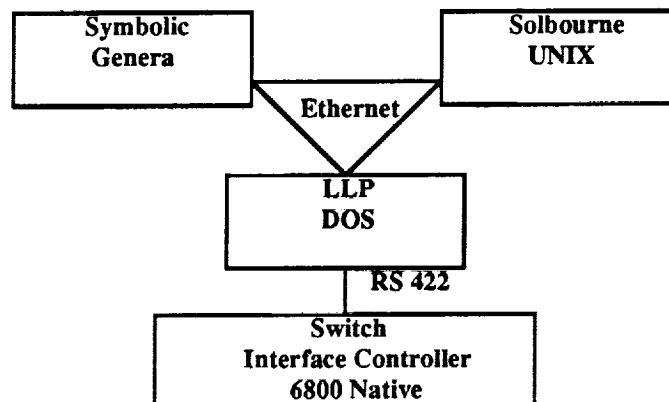


Figure 3-2 Present SSM/PMAD Systems Level Architecture

The resulting architectural advantage for the SSM/PMAD system is the co-location of automation management software (KNOMAD-SSM/PMAD and FRAMES) and multi-tasking system operational software (UNIX) on Solbourne workstation. Now the central knowledge and fault management functions within the system can take advantage of the operating system features which allow the independent processes to be individually managed.

A new software knowledge management system, KNOMAD, has been introduced into the SSM/PMAD automation environment. KNOMAD resides on the Solbourne workstation and manages the rules, objects, and databases associated with the knowledge agents. This provides a very fertile environment for cooperating expert systems both on the Solbourne workstation and between agents at the Solbourne and other workstations. Figure 3-3 depicts the KNOMAD system on the Solbourne workstation.

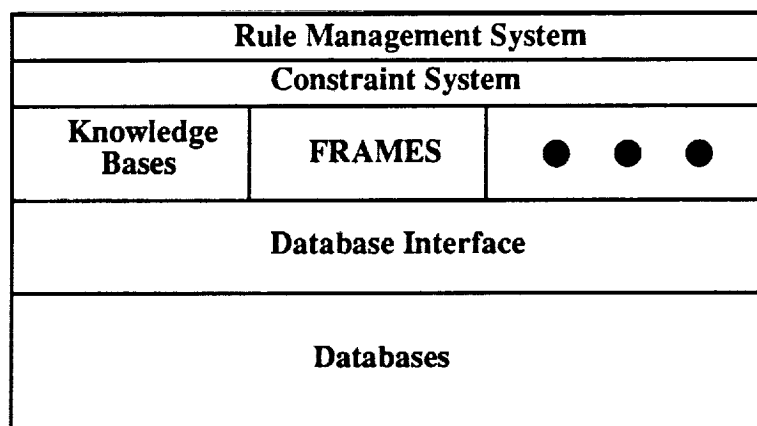


Figure 3-3 The KNOMAD System Within the SSM/PMAD

At this time only the FRAMES knowledge agent executes within the KNOMAD environment. However, the capability to include LPLMS and FELES currently exists. The presence of KNOMAD in the SSM/PMAD architecture greatly enhances the performance of knowledge bases which must interact with the data supplied both to and

from the LLPs. Perhaps KNOMAD's most important contribution in an architectural sense is the capability to manage data which affects the performance of knowledge agents in a near-real-time manner; and, to determine which knowledge agent should respond to events based upon the changing knowledge and data within the SSM/PMAD system.

The strength of this SSM/PMAD architecture is that new software introductions into the automation system should show up as enhancements to the existing elements. For example, a new expert system application would be introduced as a new knowledge agent within the KNOMAD environment; or, a change to the LLFs would only show up as a modification to be executed at the LLPs. The architecture is now highly modularized and changes are isolated to the modular elements.



4.0 FRAMES RULES ENGLISH REPRESENTATION

The following descriptions are English Language representations for the hard fault rules (diagnoses rules only; readers are referred to Appendix III for explanation of any rules used for grouping or control) that were inserted into the FRAMES Knowledge Agent which executes within the KNOMAD environment.

4.1 Hard Multiple Fault Rules

Rule 2.1 For all tripped switches

IF
the voltage of the top sensor in the hierarchical group is under-voltage,
THEN
diagnose as there is no power to the bus, and
report it to the operator.

Rule 2.2 For all tripped switches

IF
the fault is under voltage, and
the top sensor voltage is nominal, and
the switch sensor voltage is nominal, and
THEN
diagnose as a broken cable between the child sensor and switch, and
report it to the operator.

Rule 2.3 For all tripped switches

IF
the fault is under voltage, and
the voltage of the parent switch sensor is nominal, and
the voltage of the tripped switch sensor is less than nominal, and
the parent switch can trip but is not,
THEN
diagnose as a broken cable below the parent switch, and
report it to the operator.

Rule 2.4 For all tripped switches

IF
the fault is under voltage, and
tripped switches sensor voltage is less than nominal, and
parent switch is not trippable on under voltage, and
parent switch sensor voltage is nominal, and
the voltage of the parent switch sensor is nominal,
THEN
diagnose as the input or output of the switch above is broken, or
the switch itself is broken, and
report it to the operator.

Rule 2.5 For all tripped switches

IF
the fault is under voltage, and
tripped switches sensor voltage is less than nominal, and
parent switch is trippable on under voltage but not tripped, and
parent switch sensor voltage is nominal, and
sensor voltage above the parent switch is less than nominal,
THEN
diagnose as broken under voltage sensor in parent switch, and
broken cable above parent switch, and
report it to the operator.

Rule 3.2.3 For all tripped switches

IF
all symptoms are at the bottom level, and
all are fast trips, and
all switches are not being used by a common activity, and
no switches have permission to test,
THEN
take no action, and
notify the operations personnel through the user interface.

Rule 3.2.4 For all tripped switches

IF
all symptoms are at the bottom level, and
all are fast trips, and
all switches are not being used by a common activity, and
switches have permission to test, and after testing
non-isolated symptoms result,
THEN
take no action, and
notify the operations personnel through the user interface.

Rule 3.2.5 For all tripped switches

IF
all symptoms are at the bottom level, and
all are fast trips, and
all switches are not being used by a common activity, and
switches have permission to test, and after testing
no symptoms occur,
THEN
take no action, and
notify the operations personnel through the user interface.

Rule 3.2.6 For all tripped switches

IF

all symptoms are at the bottom level, and
all are fast trips, and
all switches are not being used by a common activity, and
switches have permission to test, and after testing
a single symptom occurs, and
the symptom is a fast trip, and
the fast trip occurs in one of the same switches,

THEN

identify a possible low impedance short below the switch, and
notify the operations personnel through the user interface.

Rule 3.2.7 For all tripped switches

IF

all symptoms are at the bottom level, and
all are fast trips, and
all switches are not being used by a common activity, and
switches have permission to test, and after testing
a single symptom occurs, and
the symptom is not a fast trip, or
the fast trip occurs in not one of the same switches,

THEN

identify an unexpected trip that is not presently diagnosable, and
notify the operations personnel through the user interface.

Rule 4. through 5. For all tripped switches

IF

there are multiple symptoms, and
the symptoms are individual,

THEN

group the individual symptoms, and
diagnose each individual symptom.

4.2 Hard Single Fault Rules

Rule 6.1 through 6.4

Manipulate the switches of interest by opening and flipping the top switch and collecting symptoms.

Rule 6.5

IF
 there is a single symptom, and
 it is from the same switch as before, and
 it is the same fault symptom,
THEN
 form diagnosis on retrip.

Rule 6.6

IF
 there is a single symptom, and
 it is not the same fault symptom as before, or
 it is not from the same switch as before,
THEN
 notify the user of an unexpected retrip.

Rule 6.7

IF
 there is no new symptom, and
 there are no child switches,
THEN
 notify the operator that the fault cannot now be repeated, and
 it has possibly burned clear.

Rule 6.8

IF
 there is no new symptom, and
 there are child switches,
THEN
 close the top switch in preparation for collecting fault
 symptom data from lower switches where permitted.

Rule 6.8.1

IF
 there is a new trip on the top switch,
THEN
 notify the operator of an unexpected trip.

Rule 6.8.2

IF

there is no new trip on the top switch,

THEN

it is ok to set up child switches for testing.

Rules 6.9 through 6.12 isolate testable switches and perform opening and closing of those switches. Parameters used in diagnoses are set from these switching operations.

Rule 6.13

IF

there is more than one new top symptom,

THEN

report an unexpected number of top symptoms to the operator.

Rule 6.14

IF

there is one new top symptom, and
it is the same original fault symptom,

THEN

diagnose a sensor masked fault at the tripped child switch, and
report it to the operator.

Rule 6.15

IF

there is one new top symptom, and
it is not the same fault symptom,

THEN

report an unexpected new top symptom to the operator.

Rule 6.16

IF

there is no new top symptom,

THEN

test the child switches by closing them in sequence and proceeding
down into another layer for testing.

Rule 6.17 and 6.18

IF

there are no new child symptoms,

THEN

set the child switches to the union of the children
of the present child symptoms.

Rule 6.19

IF
 there are no child switches, and
 there have been switches that are not testable,
THEN
 diagnose the fault as not found, and
 report it to the operator with the qualifier that
 some switches were not testable.

Rule 6.20

IF
 there are no more switches to test,
THEN
 diagnose the fault as not found, and
 report it to the operator.

Rule 6.21

IF
 there are lower level switches to test,
THEN
 set up appropriate variables, and
 loop back through 6. level rules.

Rule 6.22

IF
 there are new symptoms as a result of earlier switch closings,
THEN
 order and group the symptoms, and
 get the top symptoms.

Rule 6.23

IF
 there is a top symptom that is the same as the fault symptom, and
 the switch is the same as the fault switch,
THEN
 diagnose the fault as a possible over-current, and
 report it to the operator.

Rule 6.24

IF
 there is a top symptom that is not the same as the fault symptom,
THEN
 diagnose as an unexpected different trip, and
 report it to the operator.

Rule 6.25

IF

there are multiple new top symptoms,

THEN

diagnose as unexpected different multiple trips, and
report it to the operator.



APPENDIX I SSM/PMAD INTERFACE USER MANUAL



SSM/PMAD Interface User Manual

Version 1.0

Joel D. Riedesel
Martin Marietta Space Systems
P.O. Box 179, MS: S-0550
Denver, Co. 80201
jriedesel@den.mmc.com



5.0 THE SSM/PMAD USER INTERFACE

5.1 The System Environment

The SSM/PMAD user interface gives the operator a method of interacting with the testbed. The SSM/PMAD interface was designed to be a good user interface satisfying a number of conditions. First, the user interface is easy to learn and use. The relative measure of ease increases with the complexity of the task confronting the user. The system appears homogeneous to the user. Second, it gives encouragement, usually through enhanced performance and aesthetic value. Third, the SSM/PMAD interface was designed to not inhibit the performance of a user familiar with the system. Last, the user interface meets the requirements of the testbed's operation. For a complete description of the operation of the SSM/PMAD user interface see Appendix I.

5.2 The SSM/PMAD User Interface Definitions

The user interface will be monitored by the application providing access and suppleness for human interaction. The application may be broken into tasks and functions which may control parts of the user interface. Moreover, there may be more than one application requiring a user interface. Also, the user interface may contain a variety of menus, workboxes, buttons, and graphical items. Before designing a user interface, a number of terms were defined. These terms may be placed in one of two categories.

5.2.1 CONTROL STRUCTURES

Application -

An object or module which performs a complex service or action used to accomplish some major objective.

Task -

A working component of an application.

Function -

A small operation which has a specific goal and is part of some task.

5.2.2 *USER INTERFACE ITEMS*

Window -

Rectangular region on the screen which is owned and managed by an application, task, or function requiring screen I/O. In an automobile, the speedometer and tachometer would be analogous to a window on a computer screen in that they provide the driver with visual feedback on the performance of the engine.

Menu -

Selection list owned by an application, task, or function. In an automobile, there are many buttons on the dashboard controlling such things as windshield wipers, lights, and the radio. All of these choices could be considered as menu selection possibilities for specific functions of an automobile's user interface.

Workbox -

Temporary window owned by an application, task, or function. A workbox differs from the owning window in that it services a particular need or set of needs within the owning window and does not represent the complete environment. Also, a workbox, due to its nature, possesses only a limited set of user oriented information. For example, in some automobiles, the engine has a computer control mechanism which will turn on a light when the computer believes that it is time for the driver to shift gears. This light stays on until the driver actually does shift gears. Because of the temporary nature of the shift light, it could be considered as workbox in the automobile's user interface.

Buttons -

Single function screen items selectable by a single mouse click.

Graphical Items -

Images which are found in a window on the interface screen. They may be classified as one of the following:

Icons -

Items with one alternate representation. There is no analogy for this item in terms of an automobile's user interface. In our present SSM/PMAD system, any load center may be represented on the FRAMES user interface as a small box showing none of the switches or loads. Under normal circumstances, this representation is not used, but a user might switch to this representation to make the interface easier to read.

Static Representations -

Items not affected by user interaction. For instance, in an automobile, the numbers painted on an analog speedometer are for reference and are therefore static to the user.

Dynamic Representations -

Items the user may change, move, or otherwise interact with. Although the numbers painted on an analog speedometer in an automobile are static representations, the needle which points at them is a dynamic representation. The needle position is directly related to the velocity of the automobile, and is controllable by the user through the user interface.

5.3 Color in the SSM/PMAD User Interface

The addition of color in the SSM/PMAD user interface is instrumental in providing information. Color coding of the switch icons is used to inform the user of the situation on

a bus within a given load center. The user who wishes to manually power a given load or obtain system information during automated operation sees the following color coded switch icons:

RED -

This switch is tripped.

GREEN -

This switch is usable.

BLACK -

This switch is presently out of service.

By using this color coding scheme, the user is able to "drive" the user interface and with the knowledge provided by the colors have confidence in the system results. Continuing work on the SSM/PMAD interface will provide a more robust and easily understood environment for the testbed operator.

6.0 SSM/PMAD FUTURE DIRECTIONS

6.1 SSM/PMAD Testbed Needs

The technology of the SSM/PMAD testbed should be made available to complex spacecraft development programs. As well, the testbed technology itself should be upgraded to better support the known needs of such programs as the Space Station Freedom and its derivative programs.

6.1 SSM/PMAD Testbed Goals

Accomplishing the testbed needs suggests the following six future efforts as goals for the automation capability of the SSM/PMAD testbed.

First, the user interface should be incrementally improved to the next level of operational capability. This would introduce all new functionality as well as expand and enhance the existing functionality.

Next, a planning agent should be introduced and the incremental autonomy manager should be developed. This would give the operator access to system resources without demanding that the user assume full operational control. It would also serve to minimize impacts to already scheduled operation on the automation list.

Third, the LLP software should be rewritten in Ada, and as many of the FRAMES functions as possible should be rehosted to that level, taking advantage of the functionality provided by Ada.

Next, the LPLMS and FELES should be placed into the KNOMAD environment to make the user interface functions more completely homogeneous. This would also take advantage of the parallel management structure allowed by KNOMAD and would provide a more complete structure in the sense of cooperating expert systems.

Fifth, there should be a two-tiered explanation facility introduced into the SSM/PMAD. The first tier would explain the processes within the KNOMAD knowledge

management environment. The second tier would provide information on the system inter-process interactions, giving the operator a sense of what the overall system is doing.

Sixth, the KNOMAD system should be enhanced to mix the forward and backward reasoning processes in an automated fashion. This would alleviate knowledge engineering personnel from having to manually determine complete knowledge processing strategies ahead of time.

Contents

1 Purpose of this Manual	1
1.1 Acronyms and Definitions	1
2 Installation of FRAMES on the Solbourne	4
2.1 Loading the Software	4
2.2 Changing the X server	5
2.3 Editing the hosts File	5
2.4 Copying the Initialization Files	5
2.5 Creating an LLP Software Disk	6
3 Breadboard Tutorial	7
3.1 The Power System	8
3.1.1 Breadboard Power	8
3.1.2 Starting the LLPs	8
3.1.3 Shutting Down the Power System	8
3.2 The Symbolics	8
3.3 The SSM/PMAD Interface	9
3.3.1 Starting the SSM/PMAD Interface	10
3.3.2 Initializing FRAMES	10
3.3.3 Manually Operating the Breadboard	12
3.3.4 Getting Information from the Hardware	12
3.3.5 Autonomously Operating the Breadboard	13
3.3.6 Stopping FRAMES	14
3.3.7 Shutting Down the SSM/PMAD Interface	14
4 The SSM/PMAD Breadboard	15
4.1 SSM/PMAD Breadboard Theory of Operation	15
4.1.1 The SSM/PMAD Goal	15
4.1.2 Hardware and Software Functional Division	16
4.1.3 Manual System Operation	18
4.1.4 Autonomous System Operation	20
4.2 The FRAMES Knowledge Base	22
4.2.1 The FRAMES Architecture	22
4.2.2 The FRAMES Expert Systems	24
4.2.3 Multiple Faults in SSM/PMAD	24
4.3 The SSM/PMAD Interface, in Detail	25
4.3.1 The Screen	25
4.3.2 The Menu Functions	30



4.3.3 The Power System Components' Functions	51
A Known Bugs	58



List of Figures

1	SSM/PMAD Breadboard HW/SW Configuration	18
2	The SSM/PMAD Interface	26
3	The SSM/PMAD Interface with LLPs	27
4	The Initial SSM/PMAD Interface	28
5	Example One-line Information Message	31
6	The KNOMAD Menu	32
7	The KNOMAD Help Window	32
8	The Utilities Menu	33
9	The Communications Menu	33
10	The Utilities Help Window	34
11	The Communications Help Window	34
12	The Communications Windows	36
13	A Prompt Window for Displaying a Transaction	37
14	A FRAMES Transaction	37
15	Closing a Communications Connection	38
16	The Summary Menu	39
17	The Switchgear Summary Menu	40
18	The Power Utilization Options Menu	40
19	The Schedule Options Menu	41
20	The Summary Help Window	41
21	The Switchgear Summary Help Window	42
22	The Power Utilization Help Window	42
23	The Schedule Help Window	43
24	The Initialize Menu	44
25	The Initialization Options Menu	44
26	The Initialize Help Window	45
27	The FRAMES Initialization Help Window	45
28	The Other Initialize Menu	46
29	The Help Menu	47
30	The LLPs Help Window	48
31	The RPCs Help Window	48
32	The Loads Help Window	49
33	The Sensors Help Window	49
34	The Cables Help Window	50
35	The Mouse Cursor Help Window	50
36	The Main Menu Help Window	51
37	The Display Windows Help Window	52
38	An LLP Menu	53



39	An RPC Menu	53
40	A Sensor Menu	54
41	A Cable Menu	55
42	A Load Menu	55
43	Switch Data	56
44	Sensor Data	56
45	Load Data	56



1 Purpose of this Manual

The purpose of this manual is to give the user instructions and guidance in the general operation of the breadboard and specifically operation of FRAMES on the Solbourne. The Solbourne provides the primary user-interface for the operation of the SSM/PMAD breadboard known as the SSM/PMAD Interface. This interface will be the primary focus of the user manual.

The manual is organized into four sections. Section two describes the installation of the SSM/PMAD interface and FRAMES on the Solbourne. Section three provides a tutorial overview of the SSM/PMAD breadboard. The fourth section is the main part of the user manual and provides the user with the necessary detail for operating the SSM/PMAD interface. It also provides a fairly detailed description and theory of operation of the breadboard and FRAMES diagnostic system.

The rest of this section defines the acronyms and some brief definitions that are used throughout this manual.

1.1 Acronyms and Definitions

AI Artificial Intelligence. The field of Computer Science studying aspects of human intelligence and how they can be implemented on a computer.

CLOS Common LISP Object System. An object-oriented programming framework for Common LISP.

FELES Front End Load Enable Scheduler.

FRAMES Fault Recovery And Management System. FRAMES is logically defined to exist on both the Solbourne and the LLPS. Both of these parts of SSM/PMAD play an important role in detecting, diagnosing and recovering from power system faults.

GC Generic Controller. An element of the hardware of the SSM/PMAD breadboard for interfacing to and controlling RPCs.

H/W Hardware. The SSM/PMAD breadboard hardware components.

ICD Interface Control Document.

KHz KiloHertz.

KNOMAD-SSM/PMAD Knowledge Management and Design Environment applied to the SSM/PMAD domain.

LC Load Center. An element of the SSM/PMAD breadboard where loads may be connected to 1K-RPCs.

LISP List Processing. The programming language of choice for developing complex software systems utilizing AI technology.

LLP Lower Level Processor. A computer system for controlling LCs or PDCUs.

LPL Load Priority List. The load priority list is used to determine which loads are more important than others in the event of power system contingencies.

LPLMS Load Priority List Management System. The expert system for creating the load priority list from a schedule.

MAESTRO Master of Automated Expert Scheduling Through Resource Orchestration. The scheduling system for scheduling activities making efficient use of resources, including power.

MSFC Marshall Space Flight Center.

NASA National Aeronautics and Space Administration.

PDCU Power Distribution Control Unit. A power distribution unit of the SSM/PMAD hardware.

RBI Remote Bus Isolator. A remotely controllable relay for carrying large amounts of power (e.g. 15KW).

RCCB Remote Control Circuit Breaker. A relatively smart switch for switching 10KW of power.

RPC Remote Power Controller. An smart switch that may switch either 3KW or 1KW of power.

RS-423 Electronic Industries Associates RS standard for communications. The RS-423 is used for communications between the LLPs and the SICs.

SI Symbolics Interface. The interface for defining schedules for the SSM/PMAD breadboard.

SIC Switchgear Interface Controller. The element of the hardware for interfacing to a bus of switches. In the case of a PDCU, it controls all the switches (only one bus in a PDCU).

SSM Space Station Module. Ask NASA.

SSM/PMAD Space Station Module Power Management And Distribution. A testbed for trying and evaluating mechanisms for autonomously and manually operating the power system for Space Station Freedom.

S/W Software. Those elements of the SSM/PMAD breadboard functions that are not hardware.

TCP/IP Transport Control Protocol / Internet Protocol. The communications medium used between the LLPs, the Solbourne and the Symbolics.

UI User Interface. The place where the user interacts with the SSM/PMAD breadboard.

VCLR Visual Control Logic Representation. A means for representing and documenting the logical flow of an algorithm.

2 Installation of FRAMES on the Solbourne

This section describes the installation of FRAMES and the SSM/PMAD interface on the Solbourne. This includes a subsection on installing the LLP software. Installing the scheduling software on the Symbolics is discussed elsewhere. The installation of FRAMES involves loading the requisite software onto the Solbourne and copying the necessary initialization files into the user account. This installation assumes that the Solbourne is installed and running in a normal network environment and that X windows is installed. The only actual change that will be necessary to make to the existing installation of the X windows software on the Solbourne is to change the X server to one provided by FRANZ, Inc. It will also be necessary to add a keyword to the hosts file to let the FRAMES system know where the scheduler is located.

2.1 Loading the Software

The FRAMES software distribution includes `KNOMAD-SSM/PMAD`, GNU emacs, `TEX`, `pbmplus`, and the FRANZ, Inc. X server as well as Franz's Allegro Common LISP¹. The software distribution is located in the `/usr/local` directory. To load the software you must become root or have your system administrator read this and load it for you.

To load the software insert the distribution tape in the local tape drive and perform the following steps where `%` is the unix prompt:

1. `% cd /usr/local`
2. `% mt -f /dev/rst8 rew`
3. `% tar -xvf /dev/rst8`

¹A few caveats and legal disclaimers are in order here. This installation procedure is provided by Martin Marietta Astronautics Group under contract NAS8-36433 to NASA, Marshall Space Flight Center. The installation of the software is accompanied by the delivery of a Solbourne 5/501 computer owned by NASA. It assumes a generic setup of the Solbourne that includes a 60 MB swap partition and approximately 50 MB free space for the `/usr/local` partition.

NASA has bought two Solbourne 5/501 computers under contract NAS8-36433. Each has a license for Franz's Allegro Common LISP and Common Windows on X. This installation includes the Allegro Common LISP that is licensed to NASA on their Solbourne 5/501 computers, it includes FRAMES and `KNOMAD-SSM/PMAD`, developed by Martin Marietta, GNU emacs (GNU stands for GNU's Not Unix and is copylefted by the Free Software Foundation, available to anyone free), `TEX` (this version is `tex82` developed by Donald Knuth), and `pbmplus`. The X server provided by Franz, Inc. as part of this distribution is the public domain X server (Version 11 Release 3) as distributed by MIT. Of these installed items only `TEX` is not necessary and is provided as a convenience. `Pbmplus` is a public domain set of programs for converting between various picture formats such as GIF, SUN Raster, X, bitmap, etc. `Pbmplus` is in the public domain and was developed by Jef Poskanzer.

2.2 Changing the X server

The next step is to change the X server. First you must become root. Then perform the following steps:

1. `% cd /usr/bin/X11`
2. `% rm X`
3. `% ln -s /usr/local/bin/Xsun /usr/bin/X11/X`

2.3 Editing the hosts File

The next step is to edit the `/etc/hosts` file on the Solbourne. While root the following steps should be performed:

1. `% vi /etc/hosts`
2. Move the cursor to the line where the symbolics on which MAESTRO is to be run is defined. This can be done by using `j` to move down a line in the file and `k` to move up.
3. Next type `A maestro<esc>`, where `<esc>` is the escape key.
4. Finally, type `ZZ`.

2.4 Copying the Initialization Files

The final step to installing the software is to install the initialization files into your user account and edit your `.cshrc` file. At this point you should no longer be using the root account. The installation of the initialization files will install `.clinit.cl`, `.xinitrc`, and `.twmrc` into your home directory. If you already have some of these files, you may want to save them to another name and merge them together with the newly installed files. However, you should probably understand exactly what the commands in these files are doing in relationship to operating FRAMES before you do this. This is described in more detail in the technical reference section.

As yourself, type the following commands:

1. `% cd`
2. `% cp /usr/local/knomad/..?* .`
3. `% vi .cshrc`
4. `o`

5. `setenv DISPLAY unix:0`

6. `ZZ`

7. `% source .cshrc`

2.5 Creating an LLP Software Disk

Creating a new LLP software disk is easy. First, you take the LLP executable disk (part of the SSM/PMAD delivery) and make a discopy of it onto a blank 1.2 MByte floppy disk. This is done by typing (at the DOS prompt of the LLP computer):

```
diskcopy A: A: <cr>
```

The computer will respond by prompting for either the SOURCE disk or the TARGET disk. The SOURCE disk is the LLP executable disk. The TARGET disk is the new, uninitialized disk. The computer will repeatedly prompt for one disk or the other until the copy is completed.

Once the copy is made, the LLP executable disk is no longer needed. All that is required is to give the new LLP software disk an appropriate host table. Make sure the new LLP software disk is in drive A. The following command will move the proper host table to where it is required for normal operations for LLP B:

```
copy A:\tables\hosts.B A:\cmcnet\hosts <cr>
```

For LLP C the command is the same except the first argument of the `copy` command is `A:\tables\hosts.C`, the argument for LLP A is `A:\tables\hosts.A`, and so forth. Once the host table is in place, the new LLP software disk is ready for operation.

3 Breadboard Tutorial

This section describes the operation of the SSM/PMAD breadboard as a tutorial. The breadboard may be operated in one of two exclusive modes, manual or autonomous. Manual mode means that the user is monitoring the power system and opening and closing switches manually. In autonomous mode a schedule of switch operations is prepared in advance and used for controlling the power system without operator intervention. If a switch is tripped for some reason (because of a short or under voltage situation for example) the user will be notified of it in either mode via the user interface on the Solbourne. In autonomous mode, however, the fault will be diagnosed and a contingency schedule prepared for continued operations.

Another aspect of operating the SSM/PMAD breadboard is the interaction the user must undertake to setup each of the system elements for either manual or autonomous operation.

This section of the manual describes how to start each element of the SSM/PMAD breadboard. It does not address the power system hardware and supporting rack power. It does include turning on the power to the breadboard and the LLPs². The operation of the SSM/PMAD interface on the Solbourne is discussed in detail. The operation of the Symbolics computer is only briefly described. It is understood that the user has reference to other documentation describing how to install the software on the Symbolics, bring up the MAESTRO scheduler, and define a schedule (see [1]).

This section is divided into three parts. The first describes the procedure for starting the power system and the LLPs. The second describes the procedure for starting MAESTRO and defining an initial schedule. The third part describes the operation of SSM/PMAD system from the Solbourne interface. The third section is the most important in terms of operating the SSM/PMAD breadboard.

To operate the SSM/PMAD breadboard it is recommended that the operator bring up the system in the order given here, that is, the power system, then MAESTRO and a schedule, and finally the Solbourne interface. If only manual mode is to be used, starting the Symbolics may be disregarded. This procedure is not strictly necessary, and experienced operators will have a better understanding of the various dependencies built into the SSM/PMAD breadboard, either from usage or from reading the technical reference manual. However, for easy

²The power of the SSM/PMAD breadboard is a very confusing concept, as is the hardware. When I speak of hardware in this document I mean the actual hardware for RPC's, GC's, SIC's, A/D's and supporting power for controlling that hardware. If I am speaking of the hardware that makes up one of the computers—an LLP, the Solbourne, or the Symbolics—the reference will either be clear from the context or specifically indicated. Similarly, power is ambiguous. I am not concerned with the necessary power for operating the computers, this is assumed. There are two other aspects of power in the SSM/PMAD breadboard: One is the main power used by the SSM/PMAD breadboard to supply power to loads as scheduled by either the user or MAESTRO. The other source of power is that necessary to operate the hardware of the breadboard. This second source is exactly analogous to the power used to operate the computers and is also assumed in this manual.

and straightforward use, with as few problems as possible, this sequence is recommended.

3.1 The Power System

The power system aspect of the SSM/PMAD breadboard is probably the simplest component to operate. There are two important steps to starting the power system. The first step is to turn on the power to the SSM/PMAD breadboard. The second step is to turn on the the LLPs that will be operated. It is assumed that all the wiring of the breadboard is correct and that the cabling of the LLPs and power system hardware is correct.

3.1.1 Breadboard Power

Turning on the power to the SSM/PMAD breadboard is site dependent. The only substantial requirement is that 120 Volt DC power be supplied to each bus that will be operated. For bus A (controlled by PDCU A), 120 Volt DC power should be supplied to the RBI of PDCU A. Similarly for bus B.

3.1.2 Starting the LLPs

When operating the SSM/PMAD breadboard it is important to consider which LLPs are intended to be a part of the running system. In particular, it is important to decide which PDCUs will be operated. The particular load centers are easy to add later. For ease of use though it is a good idea to decide which ones will be used up from the start and turn them all on.

To start them simply confirm that the proper system disk is in each LLP (in other words, the system disk for LLP A should be in LLP A). Then turn the LLP on and let it boot.

It is possible to add an LLP to the already operating system (while in manual mode) if the new LLP is a load center. All that has to be done is simply turn it on. The FRAMES software will recognize the new LLP. If you desire to add a first or second PDCU it is recommended that you first stop FRAMES on the SSM/PMAD interface and then reinitialize it appropriately using the interface.

3.1.3 Shutting Down the Power System

Unless there is some reason not to turn off the power system, it can be shutdown by simply turning off the LLPs and the power to the breadboard.

3.2 The Symbolics

The operation of the Symbolics is technically very difficult. However, this manual assumes the user is a competent Symbolics user and knows how to bring up the MAESTRO software,

or has done so already.

The Symbolics is used to operate the SSM/PMAD breadboard in an autonomous mode of operation. It is responsible for generating a schedule of switch operations that enable a set of activities to be executed. When a contingency occurs, the scheduling software is capable of rescheduling the activities around the faulted area of the power system, possibly deleting some activities and adding others. The goal of the scheduling software, MAESTRO, is to use the available resources, power being a primary resource, in a very efficient manner.

In this version of the SSM/PMAD breadboard system that includes the lower level FRAMES functions (the LLPs), FRAMES, MAESTRO, FELES, and LPLMS, the user of the SSM/PMAD breadboard is responsible for generating the set of activities and the schedule for autonomous operation. The Symbolics interface must then be manually started by the `start` menu function on the console screen or by the `<meta>-s` keystroke sequence. This tells the scheduling software that it is now ready for autonomous operation and should wait for an initial ready message from the SSM/PMAD interface.

When operating the SSM/PMAD breadboard and moving between manual and autonomous modes of operation, the user must make sure to start and stop (kill) the scheduler software on the Symbolics each time autonomous mode is entered and exited. Thus, when starting the Symbolics, one starts the scheduling software. Autonomous mode is then entered on the SSM/PMAD interface. When the user then enters manual mode again on the SSM/PMAD interface, the scheduling software should be stopped (by choosing the `kill` menu option from the console screen or by typing the `<meta>-k` key sequence). Then, if the user desires to enter autonomous mode again, the schedule on the Symbolics should be prepared fresh and started again.

In general, for Version 1.0 of the SSM/PMAD breadboard, the operation of the Symbolics should follow the sequence: prepare schedule-start scheduler-stop scheduler.

3.3 The SSM/PMAD Interface

In this section the main interface to the SSM/PMAD breadboard, hereafter referred to as the SSM/PMAD interface, will be described. The description will be from the perspective of general operation of the breadboard, in both a manual and autonomous mode of operation. This will include initializing the FRAMES system (and thereby, the SSM/PMAD breadboard), manually operating the breadboard switches, moving between manual and autonomous modes of operation, and stopping the FRAMES system (enabling the user to subsequently shutdown the system). Getting information from switches and sensors will also be described. A detailed description of the SSM/PMAD interface will be given in the SSM/PMAD Breadboard section of this manual which will describe some of the more advanced functions that FRAMES is capable of as well as advanced features of the interface.

3.3.1 Starting the SSM/PMAD Interface

Starting the SSM/PMAD interface is very easy. It involves starting X windows, LISP, loading the programs and running the software. This sounds fairly complex, and an advanced developer will want to understand how these various parts are important to operating the interface, however, this procedure has been automated for the average user by using the mechanism of initialization files.

To start the SSM/PMAD interface the user should perform the following two steps:

1. Login to the Solbourne.
2. Type `ssmpmad` at the UNIX prompt.

The second step will start X windows and the TWM window manager. The X initialization file will start LISP. The LISP initialization file will automatically load and start the SSM/PMAD interface.

When the interface has been started the user will be notified by a message in the FRAMES status messages window. The message will state that FRAMES is ready to be initialized. This means that the interface has been loaded but is not initialized and not running in the context of the SSM/PMAD breadboard. The next step is to initialize FRAMES.

As with any window that gets popped up on the SSM/PMAD interface display, the FRAMES status messages window may be hidden from view by clicking the right mouse button while the mouse is positioned in the title bar of the window (the mouse cursor will change to a target cursor). Pop-up windows are generally used for displaying information about the SSM/PMAD breadboard and FRAMES. All pop-up windows may be manipulated while the mouse is positioned in the title bar (some pop-up windows may be manipulated while the mouse is in the window itself). Each mouse button has an associated action for the window. The left mouse button is used to both raise a partially exposed window to the top of the screen as well as update the information displayed in the window, if appropriate. The middle mouse button, when held down, allows the user to move the window about the screen. And the right mouse button, as mentioned above, allows the user to hide or close the window as appropriate. The windows will be discussed in more detail in the SSM/PMAD Breadboard section and, as appropriate, in this section for operating the breadboard.

3.3.2 Initializing FRAMES

Once the SSM/PMAD interface is loaded and running, all the parts of the user interface are in existence and usable. However most are not accessible until FRAMES has been initialized.

When first starting the SSM/PMAD breadboard, FRAMES must be initialized. Thereafter it is not necessary to initialize it again unless FRAMES has been stopped. Initializing FRAMES starts all the necessary software on the Solbourne for manually and autonomously operating the breadboard. This includes starting the necessary communications processes,

initializing the state of the power domain model, initializing the LLPs as necessary, and starting any necessary supporting processes and expert systems.

To initialize FRAMES the user will select the appropriate initialize option from the initialization menu. The breadboard may be initialized in four ways: with power to both buses, only to bus A, only to bus B, and to no buses. This allows the user to be selective about how much of the power system is intended to be operated. This is important for both manual and autonomous operations. It is recommended that the user initializes the system with both buses by default. This will insure that the SSM/PMAD interface thinks that power is being supplied to both buses. If the user then manually operates switches, the user interface will correctly display the state of power through the cables to the switches.

There are four major operations related to the initialization of FRAMES. These are (1) initialize, (2) autonomous mode (3) manual mode, and (4) stopping FRAMES. Operation (1) is exclusive with operations (2), (3), and (4). This means that if FRAMES is not initialized the user may only initialize it (or exit). Once FRAMES is initialized either operation (2) or (3) is enabled along with operation (4). In other words, operations (2) and (3) are also mutually exclusive. The breadboard may be operated in either manual or autonomous mode. When FRAMES is initialized, the user is put into manual mode. The option to enter autonomous mode will be the first choice the user will see. After entering autonomous mode the option to go back to manual mode will be enabled. At any time after initializing FRAMES the user may elect to stop FRAMES and therefore also stop the SSM/PMAD interface.

Finally, to be specific about how to initialize FRAMES the following steps should be executed:

1. Position the mouse over the initialize menu option on the SSM/PMAD interface.
2. The mouse cursor should take the shape of an 'X'. This signifies that when any mouse button is pressed an action will occur. The user should hold down any mouse button to bring up a menu of initialization options.
3. The user should then drag the mouse over the initialize FRAMES option and pull it right—off the right end of the menu. This will bring up a sub-menu of the initialization options.
4. The user should then position the mouse cursor over the option for initializing both buses and release the mouse.

When the appropriate initialization option has been selected a message indicating that FRAMES is being initialized will be displayed in the FRAMES status messages window. When FRAMES has finished initializing the FRAMES status messages window will display a message indicating that FRAMES is initialized and the user is in manual mode. The user may then proceed to manually operate the breadboard.

3.3.3 Manually Operating the Breadboard

At this point the SSM/PMAD breadboard is operating in manual mode. The necessary FRAMES processes are running and the user may proceed to manually manipulate the switches. However, there may not be any LLPs present on the user interface with switches to manipulate.

If the LLPs were started before starting the SSM/PMAD interface, then as soon as FRAMES is initialized the user will be notified that those LLPs that were started have initiated connections to Solbourne. The LLPs that have initiated connections will be represented on the interface. Any switches that are not available will be subsequently erased from the interface as soon as the LLP notifies FRAMES of their configuration.

If the LLPs have not yet been started then it is appropriate to do so at this time.

When LLPs and switches are represented on the interface, the user may manipulate the switches. To command a switch on or off the user positions the mouse over the switch to be commanded. Any mouse button may then be held down to bring up a menu of options. At this time only two options are of interest; these are the command on and command off options. To command the switch on simply position the mouse over the command on option and release the mouse. Shortly thereafter the user should see that the switch has been commanded on by the switch turning a solid green color on the interface. Any reflection of the hardware should also be obvious at this time. To command a switch off the user has only to select the command off option.

Another feature the user has access to when manipulating the switches is that if the user is interested in turning on a lower switch, say switch C05 and if the 3k-RPC above it (in this case A03) is not already on, then if the user simply commands on switch C05, A03 will automatically be turned on as well. This provides the user with more power for operating the breadboard. There are less actions that have to be taken by the user in order to produce the desired result. This feature is also enabled for turning switches off. If the user elects to turn off an upper level switch which has switches below it on, those lower switches will be turned off first, automatically.

3.3.4 Getting Information from the Hardware

The user will undoubtedly have noticed that switch information may be selected as an option when viewing the menu of switch actions. The menu of switch actions that the user brings up by holding down the mouse button on a switch allows the user to get information from a switch in a variety of formats.

The user may also bring up a menu of options for getting information from a sensor or a cable (which simply gets information from the closest sensor above it) as well as an LLP (however, the LLP information functions are not currently implemented in Version 1.0).

For switches, the user has the option of getting normal switch information which includes the switch's state, the current going through it, if it is tripped (and how) and a couple other

items. This is the normal switch data information option. It will bring up a pop-up display window. Clicking left on the title bar of the window will cause the information to be updated to new breadboard values. The detailed switch data option pops up a window that displays which bits are set in the status word of a switch. This is detailed data about the switches that is more useful to a system developer. Both of these functions are implemented by actually querying the LLP for new data to display whenever the information is requested from the SSM/PMAD interface. There is one further option the user may choose—the continuous switch information option. This option will cause a pop-up window to be displayed that displays the same data as the normal switch data option displays. However, this window will be continuously updated with new data from the LLP about the switch for every pass the LLP makes through its operation loop. This option should be used with care. It is quite possibly to overload the SSM/PMAD breadboard with network traffic and data processing simply to accommodate continuous switch information on a number of switches and sensors.

The user may similarly get information from sensors and loads. For sensors the user will be able to monitor the current and voltage of the sensor. For a load the user will only be able to monitor if it is powered or not.

3.3.5 Autonomously Operating the Breadboard

To operate the SSM/PMAD breadboard in autonomous mode the user simply has to select the autonomous mode option from the initialization menu of the SSM/PMAD interface. This option, once selected, will cause FRAMES to reinitialize the connected LLPs, initiate a session with the scheduler system on the Symbolics and start the fault diagnosis expert system.

Entering autonomous mode requires that a schedule be ready on the Symbolics computer for operating the switches of the breadboard. Initially selecting autonomous mode takes about one minute for FRAMES to set up for autonomous operations. When FRAMES is ready and the Symbolics has sent a schedule for autonomous operations to FRAMES, the user will be prompted to select how long until the start of mission on the SSM/PMAD interface. In general the user will want to start right away and should therefore position the mouse over the one minute option and click any mouse button. When this has happened the user will note that the message that FRAMES is now operating autonomously has been displayed in the FRAMES status messages window.

At this point the user needs do nothing to operate the breadboard. It will autonomously operate itself. Switches will be turned on and off according to the schedule as prepared by MAESTRO. If a fault occurs, the user interface will reflect the status of the breadboard and perform fault diagnosis to determine which switches may no longer be used because of the fault. The scheduler will then be notified of the new state of the breadboard and will perform contingency scheduling to continue efficient use of the power system. All these actions occur autonomously.

There is not much for a user to do during autonomous operations except to observe the schedule's progress and possibly monitor any interesting switch and sensor data. At any time the user may choose to go back to manual mode by selecting the manual mode option in the initialization menu of the SSM/PMAD interface. If the user does go back to manual mode, the breadboard will remain in the last state it was in in autonomous mode. It is up to the user to turn off any on switches. The user may then proceed to again enter autonomous mode after preparing a new schedule on the Symbolics.

3.3.6 Stopping FRAMES

Stopping FRAMES is an important function. In fact, to exit the SSM/PMAD interface FRAMES must first be stopped.

Stopping FRAMES is performed by selecting the stop FRAMES option in the initialization menu of the interface. This option causes all the associated FRAMES processes to be shut down. More importantly, however, is that all the communications that has occurred between FRAMES, the LLPs, and the Symbolics is logged to archival files. Additionally, all messages that have been displayed to the status window also get logged to an archival file. This is important for debugging and development purposes. It enables an accurate record to be kept of each session of SSM/PMAD breadboard.

3.3.7 Shutting Down the SSM/PMAD Interface

Shutting down the SSM/PMAD interface is also very easy. To shut down the user selects the exit option located in the menu under the KNOMAD menu item of the SSM/PMAD interface. This option exits the LISP system causing X windows to also be exited. The user will find a UNIX prompt on the screen as a result of exiting. At this point the user may simply logout or do whatever else is desired at a UNIX prompt.

4 The SSM/PMAD Breadboard

This section of the user manual describes the theory of operation of the SSM/PMAD breadboard. It also discusses in much greater detail the options available to the user using the SSM/PMAD interface. These include the various utility functions, window operations, and other functions available to the technical user.

The user is also introduced to the FRAMES knowledge base; the FRAMES domain and the rule groups making up the soft and hard fault expert systems.

4.1 SSM/PMAD Breadboard Theory of Operation

The SSM/PMAD breadboard consists of the hardware and software for efficiently managing and operating a space station module like power system. This includes the necessary scheduling of power, the control of hardware switches, and the overall control of the breadboard during normal and contingency operations. This section will describe the goal of the SSM/PMAD breadboard, the functional division of hardware and software throughout the breadboard, both manual and autonomous system operation and contingency operations.

4.1.1 The SSM/PMAD Goal

To operate spacecraft, power systems must exist to supply the energy needed for the various components and subsystems to carry out their work. Up to now, these power systems were either managed by ground personnel performing planning and scheduling for the activities to be carried out by the spacecraft, or were managed by flight crew personnel carrying out the same activities on-board the space vehicle. In either case, a priori knowledge of the initial plan did not guarantee the production of a sound, manageable power usage schedule, and the efforts of many people were necessary to complete the required iterations to produce a manageable power usage plan for a given mission profile.

In addition to this, power usage contingencies arise within practically all missions. Planning under the conditions of a contingency often does not allow for the key personnel or the time needed to complete the task in a safe manner, regarding the appropriate priorities and how they may change with respect to time and conditions. It is generally agreed that an expert who handles the management of a contingency replanning activity does so by knowing what the important system factors are, and by tracing through those factors until arriving at a safe and acceptable plan.

The primary goal of the SSM/PMAD is to autonomously provide, manage, and update as needed an appropriate, autonomously supplied power usage schedule (reflecting the needs of loads and their respective priorities), whether under nominal conditions or a contingency. This means that the loads are provided power in the best way that the automation system can provide. The line of reasoning within each knowledge processing environment of the

SSM/PMAD instills this goal, and the deterministic processing supports it. Hence, the system has one direction and one philosophy; to simply implement and support the goal.

4.1.2 Hardware and Software Functional Division

There are a number of functional operations that must be performed to meet the SSM/PMAD goal. These functional operations include schedule generation and transformation functions to enable a schedule of activities to be transformed into switch control events for subsequent execution by the switchgear, limit checking functions to make sure that the switchgear performs within allowed specification, interfaces to specify activities to be scheduled and system operation (as well as monitoring and manipulation of the power system switchgear), and diagnosis functions for detecting, isolating, and recovering from power system faults. These functions are described here:

MAESTRO The scheduler is responsible for taking a set of activities as input and, using a model of the power system topology, producing an efficient schedule for the activities as output.

FELES A schedule translator and associated functions for both interfacing MAESTRO to the rest of the SSM/PMAD breadboard and defining a list of events for turning on and off switches of the breadboard.

LPLMS A load priority list generator. Used to define priorities on switches that map to scheduled events. These priorities are then used by the SSM/PMAD system to determine what switches can be shut off to enable switches operating activities with higher priorities to remain on in contingency operations.

FRAMES The set of expert systems and deterministic algorithms for detecting and diagnosing hard faults and soft faults during autonomous operation of the breadboard.

System Operation This function is the main controlling function of the SSM/PMAD breadboard. It manages all the other functions of the breadboard.

SSM/PMAD Interface The interface is used to operate the breadboard in both manual and autonomous modes. It is used to initialize and stop the breadboard.

Symbolics Interface This interface is used to generate activities to be scheduled for execution in the SSM/PMAD breadboard as well as actual scheduling of activities.

KNOMAD-SSM/PMAD The supporting software for the FRAMES knowledge base and advanced AI programming for the SSM/PMAD breadboard.

Limit Checking Used to determine if the amount of current going through a switch is within the allowed amount.

Load Shedding This function shuts off switches in the load centers if the amount of current going through the switch is more than allowed as determined by the limit checking function. In the PDCU, the illegal use of current is reported to the user and FRAMES if appropriate.

Schedule Execution The events generated by the schedule must be executed, causing switches to be turned on and off.

Fault Reporting When a fault occurs in the power system hardware, the fault must be reported to the user and FRAMES so diagnosis and recovery may occur.

Redundant Switching If a switch gets turned off in a contingency situation and the load being powered by that switch has been declared to be important and scheduled in a redundant fashion, then the redundant switch to the load will be turned on to keep the load powered.

Fault Isolation Fault isolation must be performed by the SSM/PMAD breadboard to determine the cause of a fault when in autonomous operation.

Performance Monitoring The SSM/PMAD breadboard is a power system, and as such, power is monitored and averaged over time. The performance monitoring function allows the user to observe how the power is being utilized compared to how it was scheduled.

The SSM/PMAD breadboard hardware and software is configured as in figure 1. Each 80386 PC functions as one LLP. Each LLP is responsible for controlling either a load center or a power distribution and control unit. This involves communicating to the SIC and A/D cards of the hardware for that LLP. Each SIC is then responsible to control a set of generic cards for operating the RPCs. The LLPs are discussed in detail elsewhere.

Fault isolation is performed by software on both the Solbourne and the LLPs. Fault isolation consists of both deciding which switches need to be manipulated to get more information about a fault and for actually manipulating those switches and observing the results.

The functional operations are partitioned to hardware elements of the SSM/PMAD breadboard for control, performance, and logical reasons (see [5]). Those operations that need to be performed quickly (within seconds) are partitioned to the LLPs. Fault diagnosis and system control operations are partitioned in the next level up and operate in the seconds to minutes region. Schedule generation and regeneration (in a contingency situation) are partitioned at the highest level of the system and operate in the minutes to tens of minutes region. Finally, the critical operations of tripping a switch due to low-impedance shorts in the

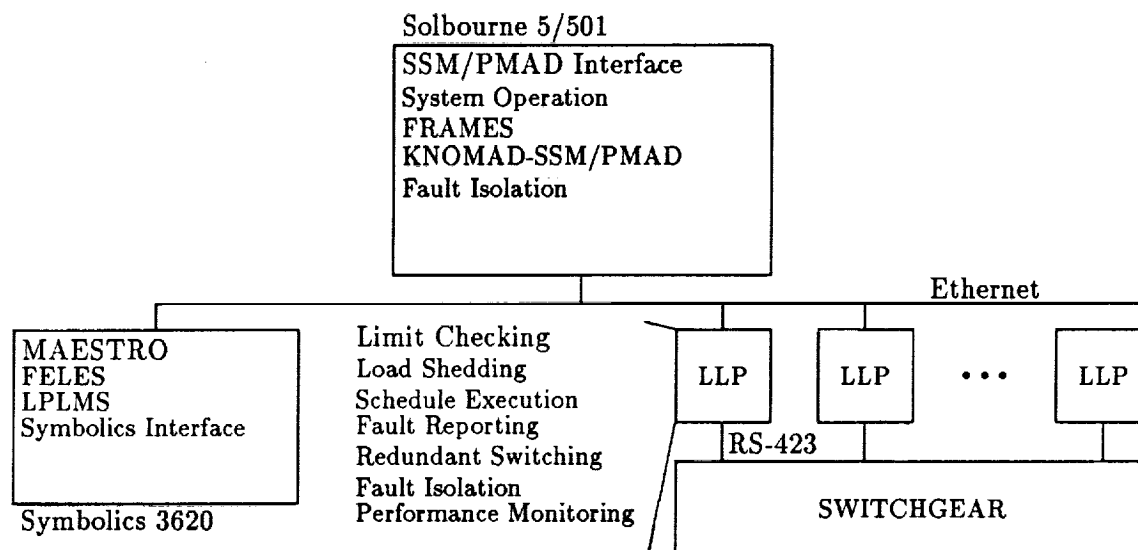


Figure 1: SSM/PMAD Breadboard HW/SW Configuration

power system are carried out in the millisecond region and are performed in the switchgear itself.

Thus, those operations that are important for safing the system remain, as in a traditional power system. What has been added are the necessary sensors and power system remote operations for the capability of installing intelligence to the power system operation and allowing the user of the system to concentrate on more important tasks such as the carrying out of experiments.

4.1.3 Manual System Operation

When the system is in manual mode, the elements of the Symbolics and FRAMES diagnostic systems are not present. Normal manual operations consists of a single interface, the SSM/PMAD interface for operating and controlling the power system switchgear. The transactions that occur between the System Operation function and an LLP are as follows during initialization:

1. An initialization message is sent to the LLP.
2. The LLP responds with three messages about its configuration.
 - (a) The switch and sensor configuration of the LLP.
 - (b) The switch conversion values of the LLP used to convert amperage readings from the switch to amps. These values may be adjusted for calibration.

- (c) The sensor conversion values of the LLP used to convert sensor readings to the appropriate units and values. These values may be adjusted for calibration.
3. The Solbourne also sends a time synchronization message to the LLP so that the LLP may initialize its clock.
 4. If the SSM/PMAD breadboard was initialized with any buses then the appropriate PDCUs will be commanded to turn on their switches.
 5. If the above step occurs for an LLP, the LLP commanded will also send a switch performance message to the Solbourne.

After the LLPs have been initialized, the user may command a switch or request data from an LLP. If a switch is commanded the following transactions occur:

1. The Solbourne sends a switch control list to the LLP to command the switch.
2. The LLP sends a switch and sensor configuration transaction back to the Solbourne to reflect the new state of the switches as a result of the command.
3. The LLP also updates its performance data on the switches and sends performance data to the Solbourne.

If a request for data is initiated the following transactions occur:

1. The Solbourne sends a query for data transaction to the LLP.
2. The LLP responds with a transaction that fulfills the request.

Every five minutes the LLPs update their performance data for switches and sensors. This information is then sent in two transactions to the Solbourne:

1. The LLP sends a switch performance transaction to the Solbourne.
2. The LLP sends a sensor performance transaction to the Solbourne.

Finally, if a fault occurs in the power system switchgear, the following transactions occur:

1. The LLP detecting the fault sends a fault transaction to the Solbourne.
2. The Solbourne then sends a quiescent transaction to all the LLPs. This transaction is a request for the LLPs to send a quiescent is true transaction back to the Solbourne when their data is quiescent.
3. Each LLP sends a quiescent is true transaction to the Solbourne.

4. The Solbourne then sends a request for switch status information to each LLP.
5. Each LLP responds with the switch status transaction.

Manual system operation is the default mode of the SSM/PMAD breadboard. It is used for manually operating the switches and observing the operation of the power system switchgear, perhaps for the purpose of calibrating the switchgear as well as testing new components. It can also be used when autonomous operation is more than necessary for the purpose of operating a device using the power system.

4.1.4 Autonomous System Operation

In autonomous system operation the addition of the FRAMES fault diagnosis software and the functional elements residing on the Symbolics are added to the manual mode software. System operation is almost identical except for initialization and contingency operations.

During initialization the following system transactions occur:

1. The Solbourne sends a ready transaction to the Symbolics.
2. The Solbourne sends an initialize transaction to each LLP.
3. The Symbolics responds with a schedule of events and a load priority list.
4. Each LLP responds with a switch and sensor configuration transaction as well as switch and sensor conversion values.
5. If the SSM/PMAD breadboard was initialized with any buses then the Solbourne will command the PDCUs to turn on their switches if they control the bus specified by the user.
6. If a PDCU had its switches turned on it will send a switch performance transaction to the Solbourne.
7. The Solbourne loads and initializes the fault diagnosis expert system software.
8. The Solbourne then prompts the user for a starting time to start the schedule.
9. The start of mission time is then sent to both the Symbolics and the LLPs to initialize their clocks.

Normal autonomous operations then include the normal periodic switch and sensor performance transactions sent from the LLPs to the Solbourne. However, in addition to these normal transactions the following additional transactions may occur:

1. Every five minutes the Solbourne will compute the utilization of power of the bread-board for each switch and for the overall power usage. This utilization data will then be sent to the Symbolics for reference purposes.
2. Periodically, FELES and LPLMS will send updated versions of the schedule and load priority list to the Solbourne.
3. The Solbourne will then distribute the various parts of these lists to the respective LLPs.

As in manual operations, the user may query the LLPs for data at any time.

When a fault occurs in the power system switchgear the sequence of transactions and operations is much more involved:

1. When an LLP first detects a fault in the switchgear it sends a fault transaction to the Solbourne.
2. The Solbourne then sends a quiescent transaction to each LLP.
3. Each LLP responds with a quiescent is true transaction when their data is in a stable state.
4. The Solbourne then sends a query for switch status information to each LLP.
5. Each LLP then sends the switch status transaction as requested.
6. The Solbourne then checks to see if the snapshot of the power system it has just collected is stable. If an LLP has had new fault data during the data collection stage it will set a non-quiescent bit in the switch status transaction. If the snapshot is not stable the Solbourne will repeat steps 2-5 until a quiescent snapshot is achieved.
7. At this time the Solbourne sends a contingency start transaction to the Symbolics.
8. The Solbourne also collects the symptoms from the snapshot and initiates the fault diagnosis expert system to perform the fault diagnosis.
9. The fault diagnosis expert system may need to manipulate the switches to gain more information about the fault (to isolate it). This process involves sending a switch control transaction to each of the LLPs with switches that need to be turned on or off.
10. The commanded LLPs will respond with switch and sensor configuration transactions to keep the Solbourne updated as to the state of the power system.
11. The Solbourne will then repeat steps 2-6 each time a fault isolation step occurs.

12. When the fault diagnosis expert system concludes a diagnosis, any switches determined to be unusable are taken out of service. Any out of service switches, load sheds, and switches switched to redundant are communicated to the Symbolics.
13. Steps 8-12 are then repeated if there are further faults (represented by symptoms not accounted for by the diagnoses so far). When all the current faults are isolated the Solbourne sends a contingency end transaction to the Symbolics.
14. MAESTRO then performs contingency scheduling, taking activities affected by out of service switches off the schedule and possibly adding other activities to the schedule.
15. The new contingency schedule and new load priority list are sent to the Solbourne.
16. The Solbourne distributes the schedule and load priority list to the LLPs.

Autonomous system operation is much more complex due to the necessity of finding out what fault really occurred in the power system switchgear and the need to update the existing schedule so that an efficient use of the power system may be made with the remaining switches. See [4] for a survey of the fault diagnosis problem and [2] for the solution to this problem for the SSM/PMAD breadboard.

In terms of actual LLP and switchgear operations, autonomous mode operations look almost identical to manual mode operations. All that is really done is to add a lot of reasoning mechanisms to the upper layers of the SSM/PMAD breadboard in order to more intelligently maintain control of the power system without requiring user intervention.

4.2 The FRAMES Knowledge Base

This part of the FRAMES tutorial describes the expert systems part of the SSM/PMAD breadboard that make up the fault diagnosis aspect of the system.

The FRAMES knowledge base is a very complex part of the SSM/PMAD breadboard. It logically consists of a domain file that defines all the data that the expert systems use to reason about the power system switchgear. It consists of domain functions used by the expert systems to compute algorithmic functions. It also consists of the main knowledge base and the various rule groups of the expert systems. These are presented in their entirety in the SSM/PMAD Technical Reference. The section of the document will discuss the FRAMES architecture and multiple faults to give a better idea of how the expert system is put together.

4.2.1 The FRAMES Architecture

The FRAMES system is partitioned into three major divisions based upon the response time needed at the different partitions. The three partitions are the distributed lower level

processors for controlling the hardware, the fault isolation and diagnosis expert systems, and the scheduling system.

The switch hardware is controlled by the lower level processes which reside on PC clones. The algorithmic processes at this level control the operation of turning switches on and off as well as monitoring power levels and performing limit checking. The lower level processes detect fault conditions which include a switch physically tripping off due to an over current or under voltage situation in the hardware. These *fault symptoms* are communicated to the fault isolation and diagnosis expert systems. Additionally, scheduled operations may no longer be performed on the tripped switches.

The response time of the lower level processors is necessarily fast. Typically, limit checking operations to shut a load off if it is using too much power, for example, are done within a one second period. The speed of the lower level processes also has implications on later fault isolation. It is quite possible that if an I^2t short is occurring in the hardware at a level of approximately 120% of the switch's rating, that it could take the switch up to five seconds to trip. The lower level processor will shut the switch off much sooner.

The third partition, the scheduling system, is not required to be nearly as responsive. Its role is to create a schedule for operating the switches in advance and to maintain that schedule during contingencies in the power system. In the present system, schedules are shipped to the lower level processors in thirty minute blocks. This allows for a semi-graceful degradation of the overall system performance if the scheduler becomes inoperable for some reason. When a fault has been diagnosed in the power system and a set of switches has been determined unusable, the scheduler is expected to reschedule its activities in a reasonable amount of time. The scheduler has been partitioned at the highest level and is expected to perform in a period of minutes.

The second partition is the fault isolation and diagnosis part of FRAMES. This part consists of a number of traditional expert systems for diagnosing different types of problems in the power hardware as well as maintaining other knowledge intensive states such as the load priority list. Currently three expert systems are defined to exist at this level: the Load Priority List Management System (LPLMS), the fault diagnosis expert system, and the soft fault expert system. However, not to confuse the issue, the LPLMS does exist at this middle layer and is currently implemented on the Symbolics.

The fault isolation and diagnosis expert system requires many supporting functions in addition to the rules that make it up. Embedding knowledge intensive applications into real world complex systems require many parts for a successful system (see [3] for one way to deal with this). In the FRAMES system these additional functions include detecting and monitoring the hardware (done by the lower level processors); communication algorithms for communicating with the distributed processors; algorithmic processes for logging data, updating database values, and the like; and user interface functions to make the system useful.

A result of this modular organization of functions in both inter and intraprocessors is

that the expert systems for fault isolation and diagnosis do not need to be executing for a person to use the system. Another way to look at it is that operating the system in an autonomous fashion requires the reasoning processes as embedded in the expert systems, while operating it manually does not.

See the paper by Riedesel, et. al., for a detailed description of the SSM/PMAD project [5].

4.2.2 The FRAMES Expert Systems

The expert systems that reside in the middle partition were mentioned above: LPLMS, fault diagnosis expert system, and soft fault expert system. These expert systems make up the FRAMES *knowledge agent*. To make FRAMES smarter, another expert system may be added to perform some useful function. These expert systems all perform in parallel with one another as if there were three people in the same room, each looking at one of the reasoning problems (LPLMS, fault diagnosis, and soft faults).

4.2.3 Multiple Faults in SSM/PMAD

The problem of multiple faults in SSM/PMAD can be divided into two cases:

Case 1 *Faults that occur within Δ time of one another.*

Case 2 *Faults that occur at least Δ time from one another.*

Where Δ is defined as: *The amount of time it takes for a fault to be initially detected and subsequently diagnosed.* Faults that occur at least Δ time from one another were already handled in the first generation of FRAMES. Faults that occur within Δ time of one another are the focus of this section.

Suppose first that multiple faults have occurred in the power system during the detection of the faults. By the time the power system has reached a quiescent state, the lower level processors will report a set of symptoms indicative of more than one fault. The fault isolation software is then tasked with determining how these collected symptoms might indicate multiple faults.

There are three cases that may be identified. The multiple faults may occur on the same bus, they may occur in the same hierarchy, and they may occur on completely independent buses. For multiple faults that occur in the same hierarchy it is possible that one of the faults could be masked (by a bad current sensor, for example) and appear to be multiple faults on the same bus.

To adequately deal with multiple faults the set of symptoms that the LLPs report are first analyzed and organized into clusters. A cluster of symptoms is a set such that each symptom in the set either occurred on the same bus or occurred below another symptom.

This leaves two cases of multiple faults for the expert system to deal with. Each cluster may be dealt with independently.

Given a cluster of symptoms, the first thing that is checked is if all the top symptoms (those symptoms of the set that are all at the highest level – all on the same bus therefore) are under voltage. If this is the case it is possible that there may be no power to the bus. To check this, the top sensor of the bus as well as various sensors above the tripped switches are looked at to see if they have nominal voltage or less than nominal voltage.

If all the top symptoms are either fast trip or over current and they all have loads hooked up that are related to the same activity, it is possible that the particular activity may be involved in the trips. If all the top symptoms are fast trip, not related by an activity and do not have any switches below them, then it is possible that one of the switches had a short below it and the other switches may have fast tripped due to energy storage (but unlikely).

Finally, if none of the above cases apply, each of the top symptoms is diagnosed as an independent fault indication. Each top symptom will be either fast trip or over current (the under voltages were diagnosed earlier). The particular top switch and the switches below it may then be tested and diagnosed as an independent fault. Now, if the fault is found somewhere below the top switch (due to a masked fault), there may have been other faults in that hierarchy. If there were, the highest (in the topology of switches) of these other faults, below the top symptom yet across from the switch finally diagnosed as the position of the fault, may also be diagnosed as independent faults.

An added complication is that the isolation and diagnosis phase is part of the Δ time. This includes commanding switches on and off in an effort to repeat the symptoms. If another fault occurs during this testing, the data collection algorithms must be smart enough to incorporate any new symptoms correctly into the existing symptoms.

4.3 The SSM/PMAD Interface, in Detail

This section of the FRAMES tutorial discusses the SSM/PMAD interface. There are three different types of mechanisms provided on the SSM/PMAD interface that may manipulate the SSM/PMAD breadboard. These are the menu functions, the options available by pressing a mouse button while it is positioned over the various components of the representation of the power system on the SSM/PMAD interface, and the pop-up display windows used for displaying SSM/PMAD breadboard information.

The SSM/PMAD interface screen will be described first. The menu functions available to the user will be described as will the options available by pressing a mouse button on a power system component. The pop-up display windows will be described in the context of their activation by the other functions available on the interface.

4.3.1 The Screen

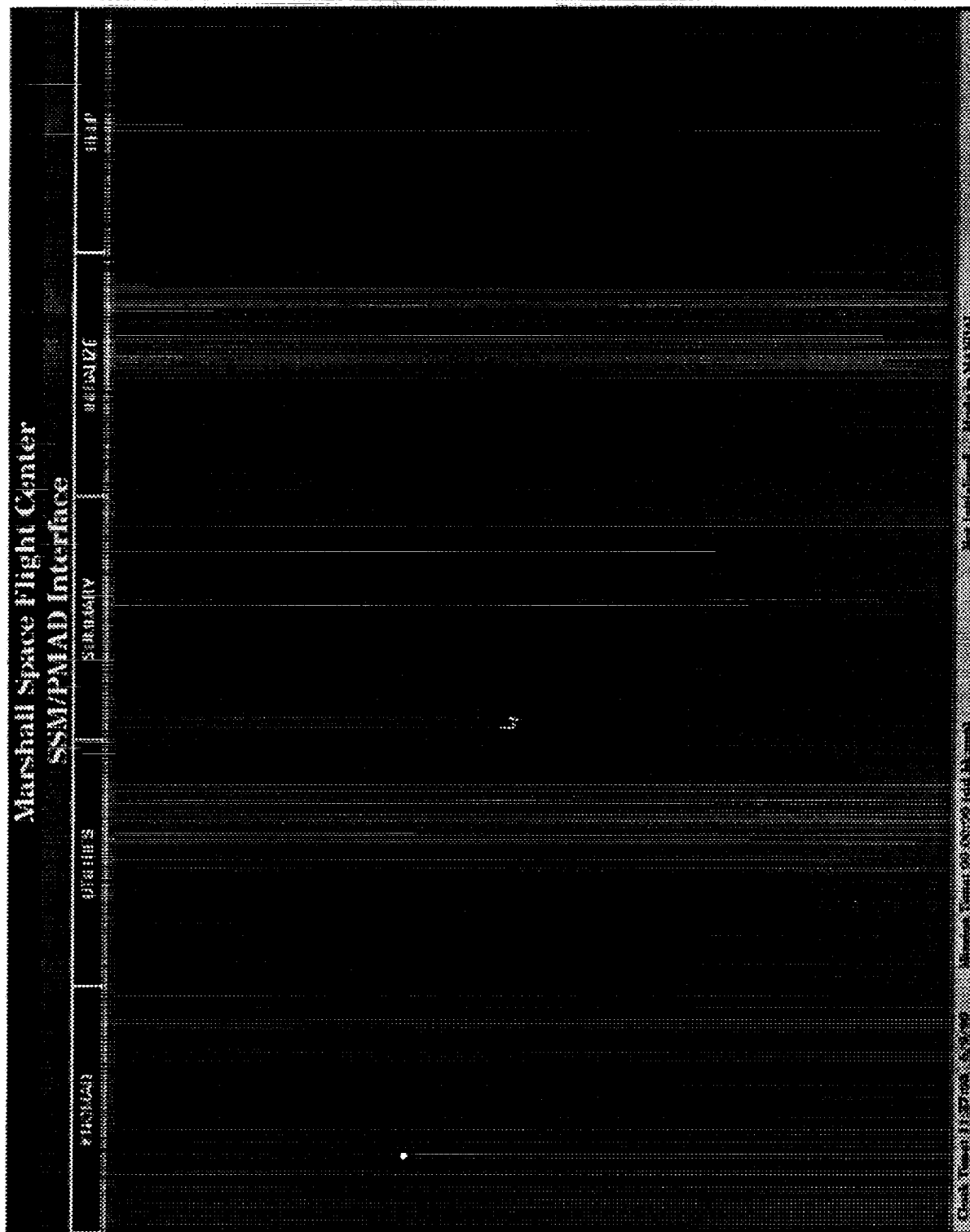


Figure 2: The SSM/PMAD Interface

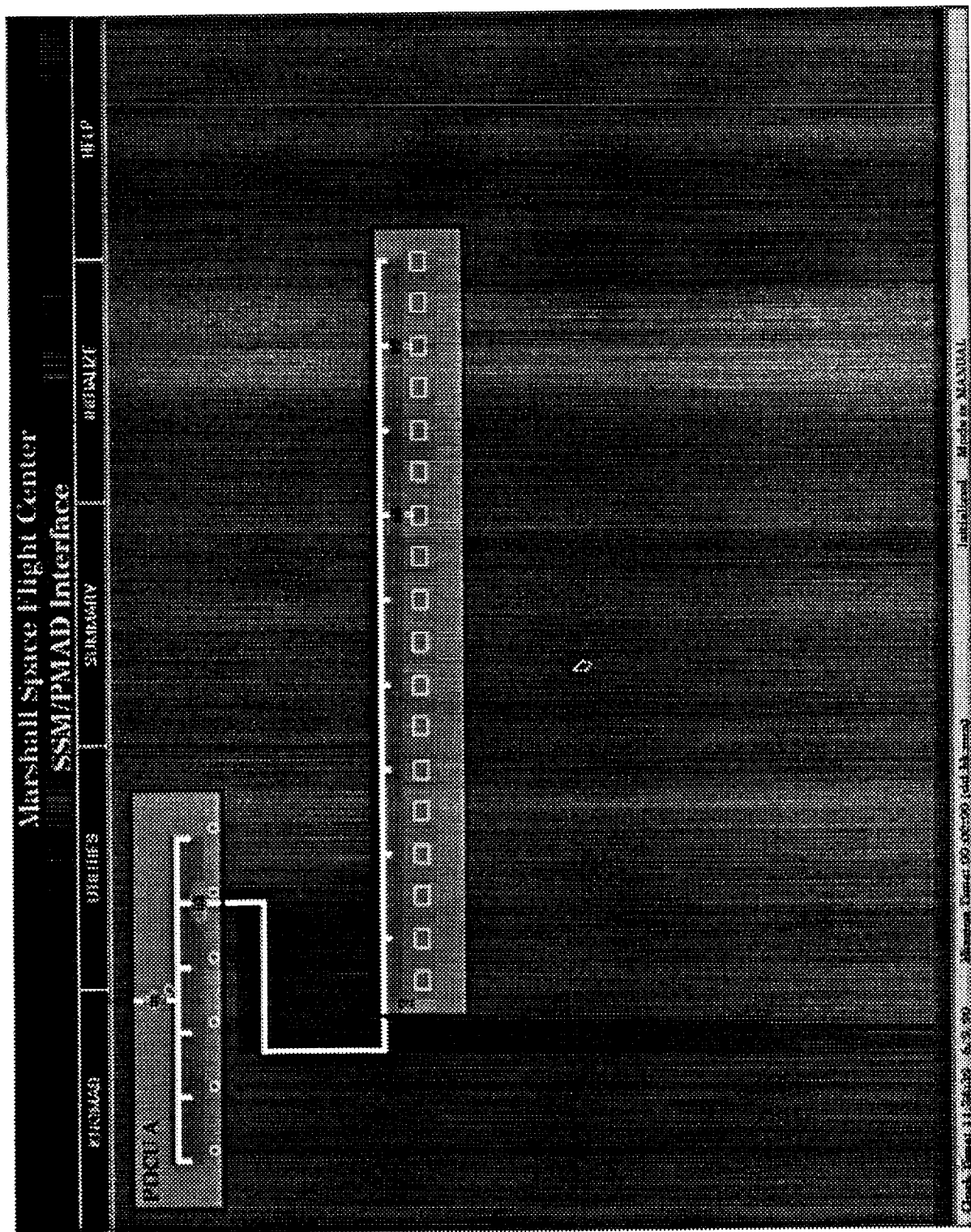


Figure 3: The SSM/PMAD Interface with LLPs

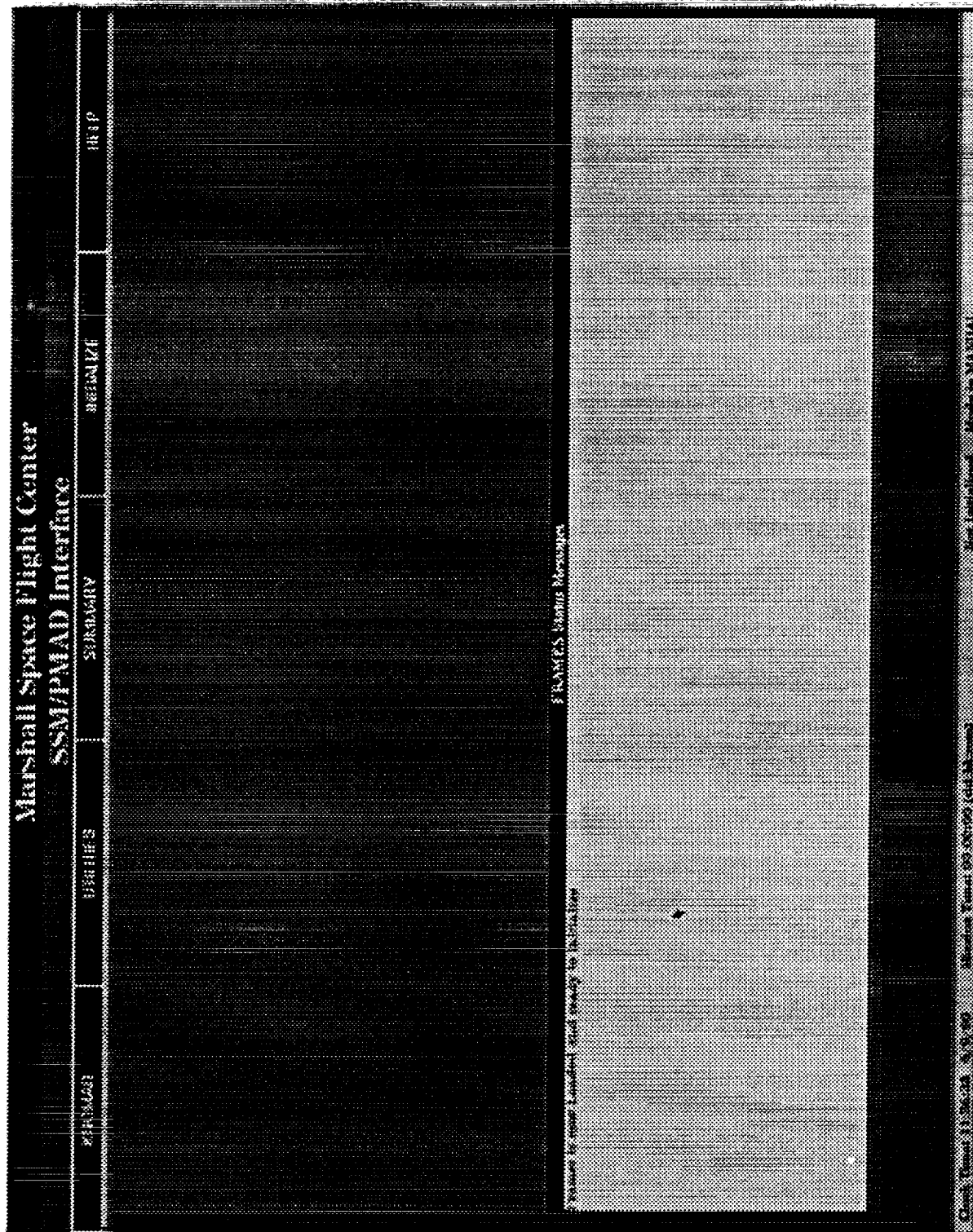


Figure 4: The Initial SSM/PMAD Interface

The main screen of the SSM/PMAD interface is shown in figure 2³. There are four functional areas to the interface. The top of the screen contains the title: *SSM/PMAD Interface*. The second portion of the screen contains the menu bar. This portion allows the user to access various functions of the interface such as utility functions, summary functions, initialization, and exiting. The third portion of the screen is the main window for viewing a representation of the power system breadboard. Figure 2 shows this area is blank. When LLPs are present in the system they appear in this portion of the screen. Figure 3 shows the interface with two LLPs present (PDCU A and Load Center 2). The user may access and manipulate the power system through its representation on the interface. The fourth and final portion of the screen is the status bar at the bottom. This bar indicates to the user information about the state of the SSM/PMAD breadboard.

The status bar represents four informational aspects of the system to the user. The current time and date is displayed first. The mission time is displayed after that. The mission time is an indication of how many minutes the schedule is into its operation. All schedules start at time zero. The mission time is only relevant when the SSM/PMAD breadboard is operated in autonomous mode. The next item is whether or not the SSM/PMAD breadboard has been initialized. The last item is the mode of operation. The breadboard always starts in manual mode.

Figure 4 shows the SSM/PMAD interface as it is first brought up. The only difference is the FRAMES status messages window in the middle of the screen. The FRAMES status messages window is used by the SSM/PMAD interface to let the user know about important activities happening in the SSM/PMAD breadboard. When the user first brings up the system the user is notified that the system has completed its initializations and is ready for general system initialization.

The FRAMES status messages window is a general purpose pop-up display window. Other pop-up display windows include the help windows of the interface. There are also more specialized pop-up display windows that are used for displaying component data; these more specialized windows do not have the capability of scrolling while the general ones do. The easiest way to find out if a window is scrollable is to move the mouse cursor slowly past the left side of the window. If the window is scrollable a scroll bar for the window should appear. The FRAMES status messages window is scrollable (as are the help windows).

All pop-up display windows have special mouse activated functions attached to them. If the mouse is positioned in the title bar of a display window, the user may choose one of three actions depending upon which mouse button is clicked. The left mouse button is defined to bring the window to the top of the screen (if it is partially exposed) and to update the contents of the window if appropriate. The middle mouse button is defined to let the user

³We realize that these screen images are not entirely readable. The problem is inherent in reducing an 8 bit deep color image to a single bit for a black and white picture. We will do our best to make sure that each item on the screen images is explained so that the context of the screen description should provide enough information to parse the especially difficult to see screendumps.

move the window. To move it, the user would hold down the middle button and drag the window to the desired location. The right mouse button is defined to hide the window from view.

4.3.2 The Menu Functions

The main user interface to the SSM/PMAD breadboard is through the functions available on the representation of the power system. However, there are a number of important functions the user has access to via the menu bar of the SSM/PMAD interface. These include various utility functions, data summarization options, initialization and exiting functions, and various help displays. Each of the menus and their options will be discussed below.

Every menu the user accesses on the SSM/PMAD interface has a help option as one of the menu alternatives. This help option will always display a context sensitive help to the user. Additionally, if the user holds the mouse over one of the menu options, a one-line message will be displayed in the small messages window of the interface. This can be seen in figure 5.

The KNOMAD Menu The KNOMAD menu is shown in figure 6. It has two options available to the user. The user may either exit the KNOMAD system or get help about the menu.

Exiting KNOMAD means that the user wishes to exit the LISP session and get back to a UNIX prompt. The exit option is only available while the SSM/PMAD breadboard is *not* initialized. If the breadboard is initialized the user will not be allowed to complete the exit option.

The KNOMAD help option displays a window shown in figure 7 that the user can read to get information about the KNOMAD menu.

All help windows are scrollable pop-up display windows. The functions available to the user on a scrollable pop-up display window were discussed above. These include scrolling the window, bringing the window to the top of the screen (if it is partially exposed) and hiding the window.

The Utilities Menu The utilities menu is shown in figure 8. The user may access three functions from this menu including a help option. The utilities help menu is shown in figure 10.

A screendump may be taken of the SSM/PMAD interface at any time by selecting the screendump option. This option will cause a compressed screendump file to be written to the user's /knomad-archive directory. This screendump will have a file name that starts with *Screen-*, followed by the date and time of the dump followed by the .Z suffix. The screendump option dumps the entire screen to this file.

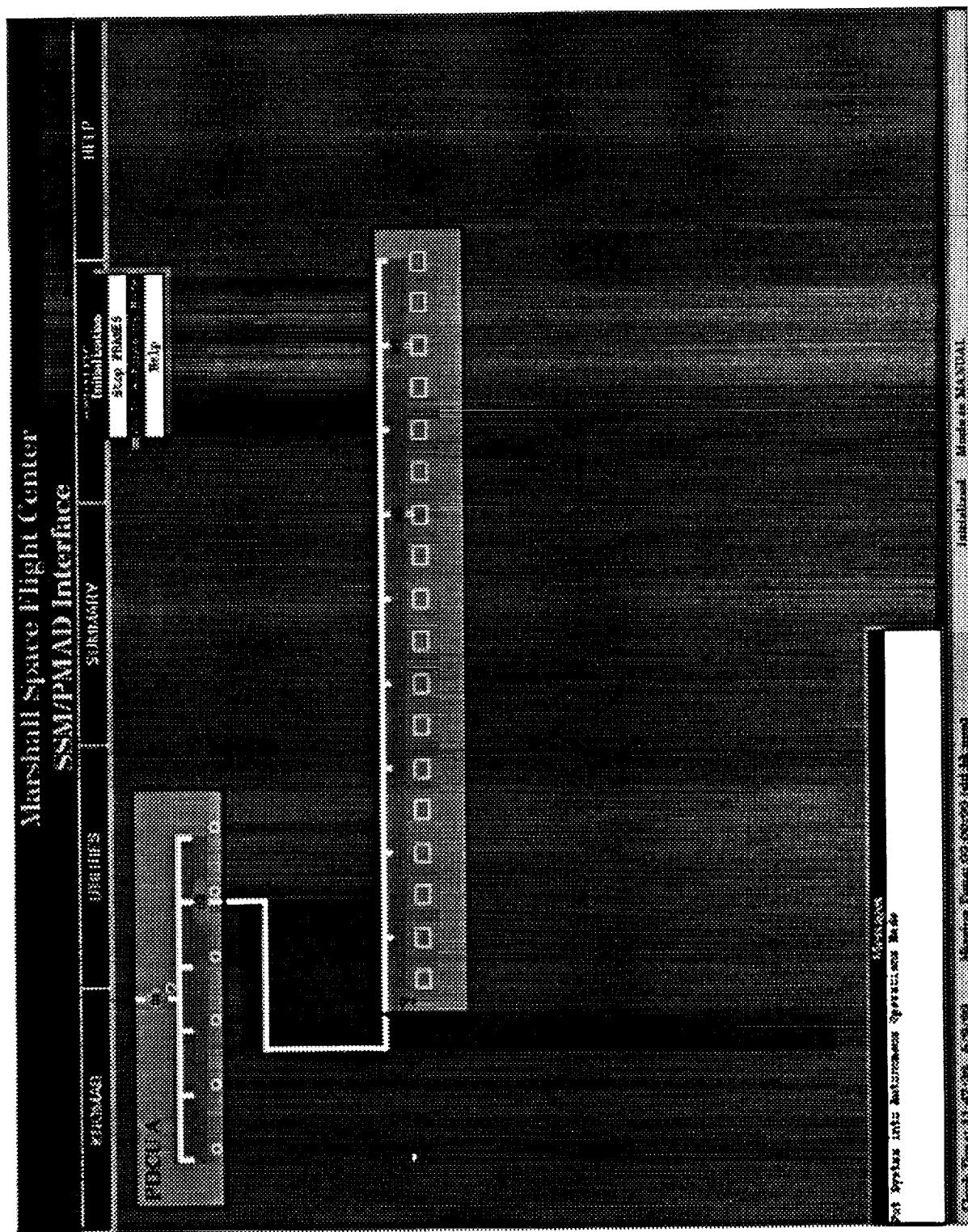


Figure 5: Example One-line Information Message

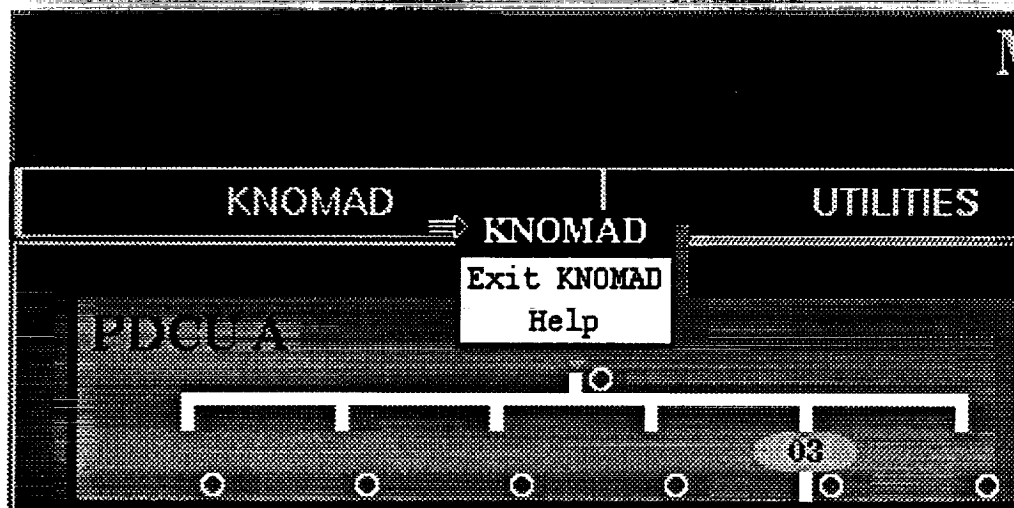


Figure 6: The KNOMAD Menu

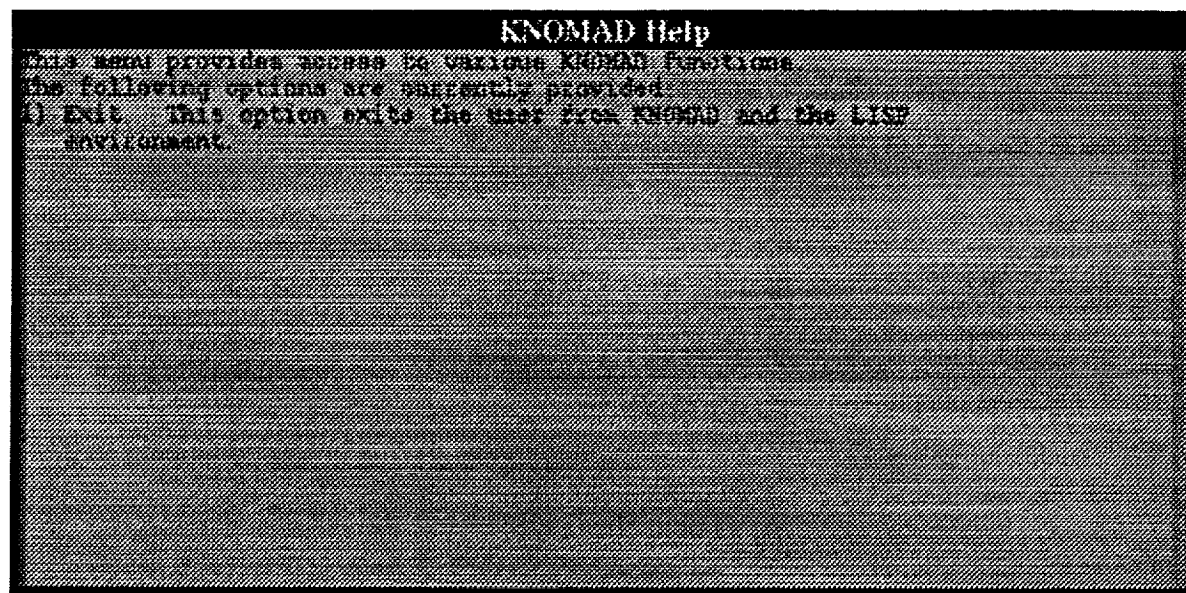


Figure 7: The KNOMAD Help Window

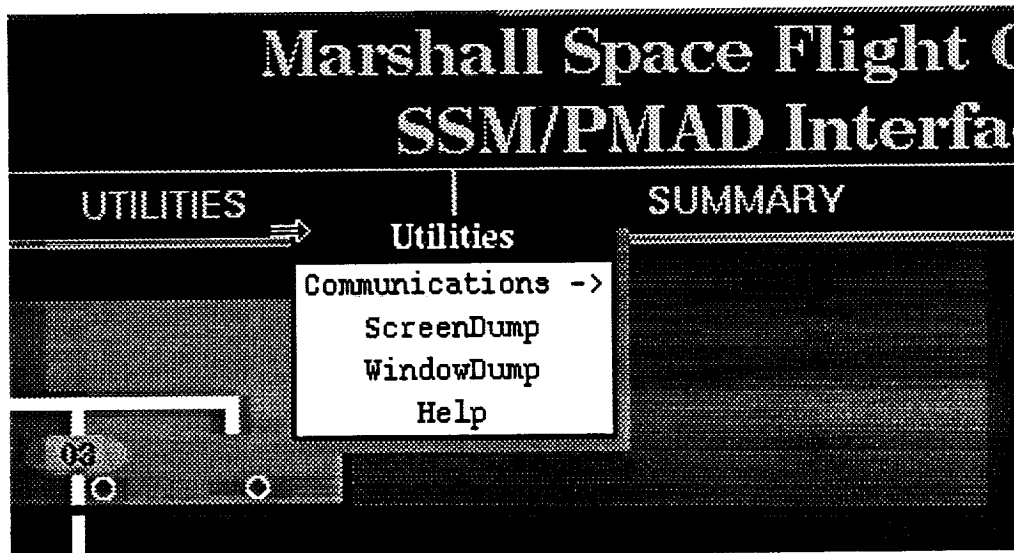


Figure 8: The Utilities Menu

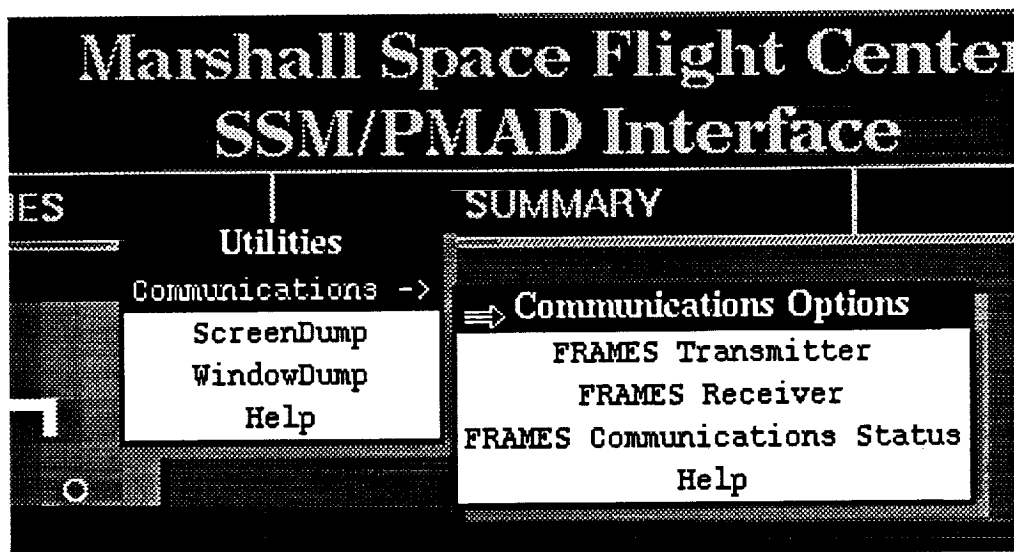


Figure 9: The Communications Menu

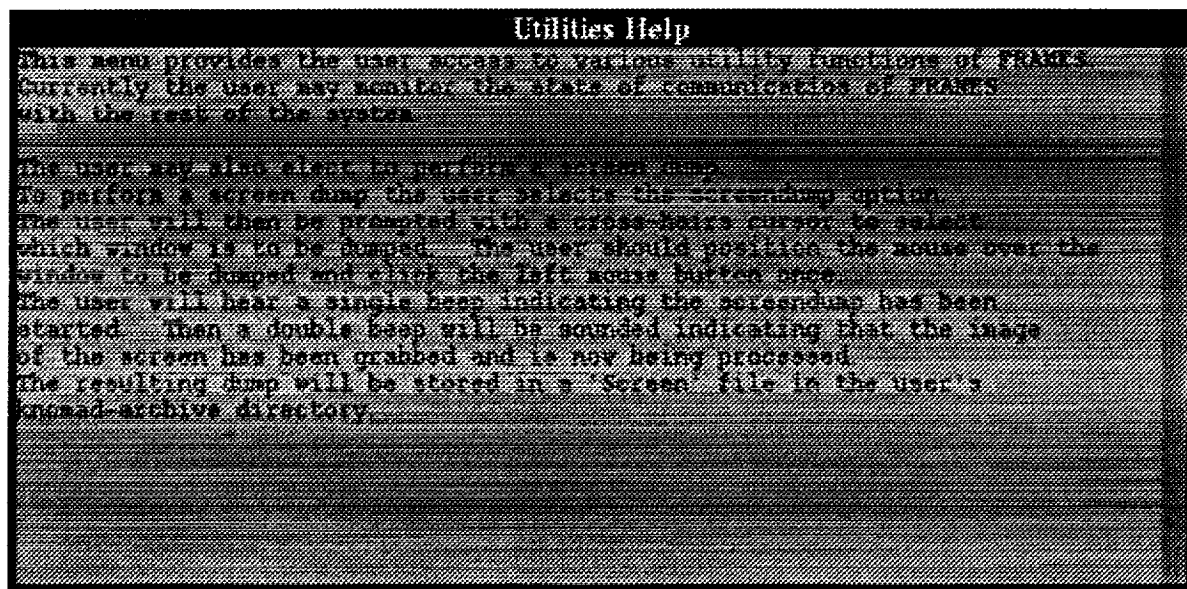


Figure 10: The Utilities Help Window

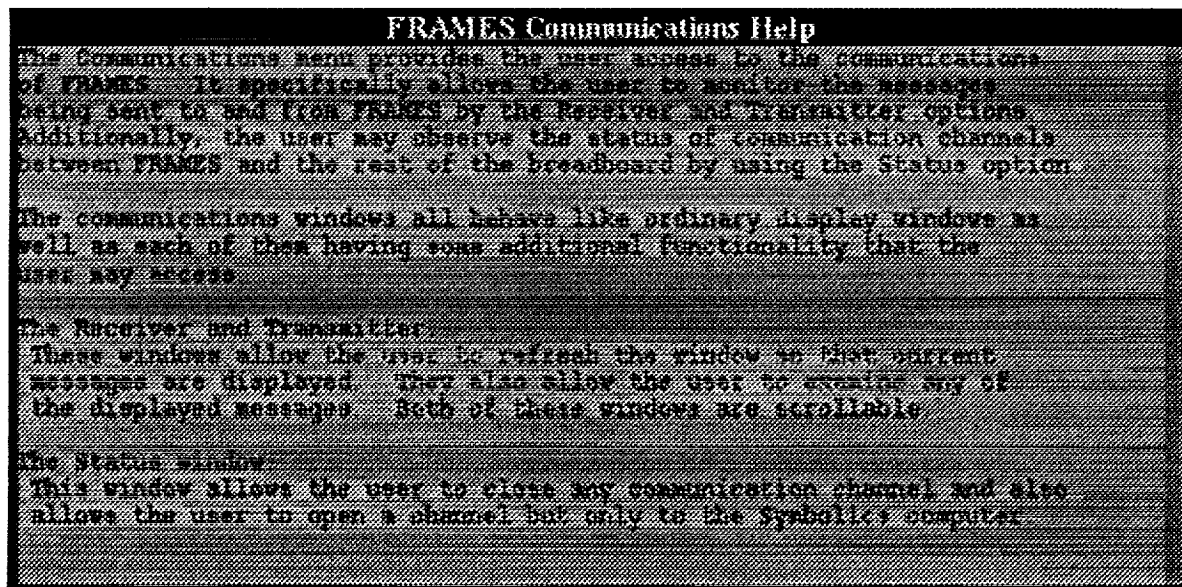


Figure 11: The Communications Help Window

A window dump may also be selected by the user. The window dump option allows the user to write only an image of a selected window to a file. As mentioned earlier, the SSM/PMAD interface is divided into four parts. Each of these parts is a separate window and may be individually dumped. Any other windows may also be dumped including the FRAMES status messages window, any pop-up display windows, etc. To perform a window dump the user selects the window dump option. When the system is ready for the dump (a couple seconds later) the mouse cursor will change to a cross-hair. At that time the user should position the cursor over the window to be dumped and click any mouse button once. The system will then beep once to indicate the start of the dump and beep twice to indicate that the image has been captured from the screen and is being processed to a file.

A window dump is also saved in the user's /knomad-archive directory with a file name of the form *Window-* followed by the date and time of the dump and a .Z as the suffix. Window dumps are also compressed.

Both window dumps and screen dumps are saved in a compressed format. If a screen dump was not compressed each one would be one megabyte. Window dumps can quickly grow large as well. The compressed format allows many window and screen dumps to be performed without having major disk space problems⁴.

The other function available to the user is a pull-right option that allows access to various communications functions of the SSM/PMAD breadboard. These options are described in the next paragraph.

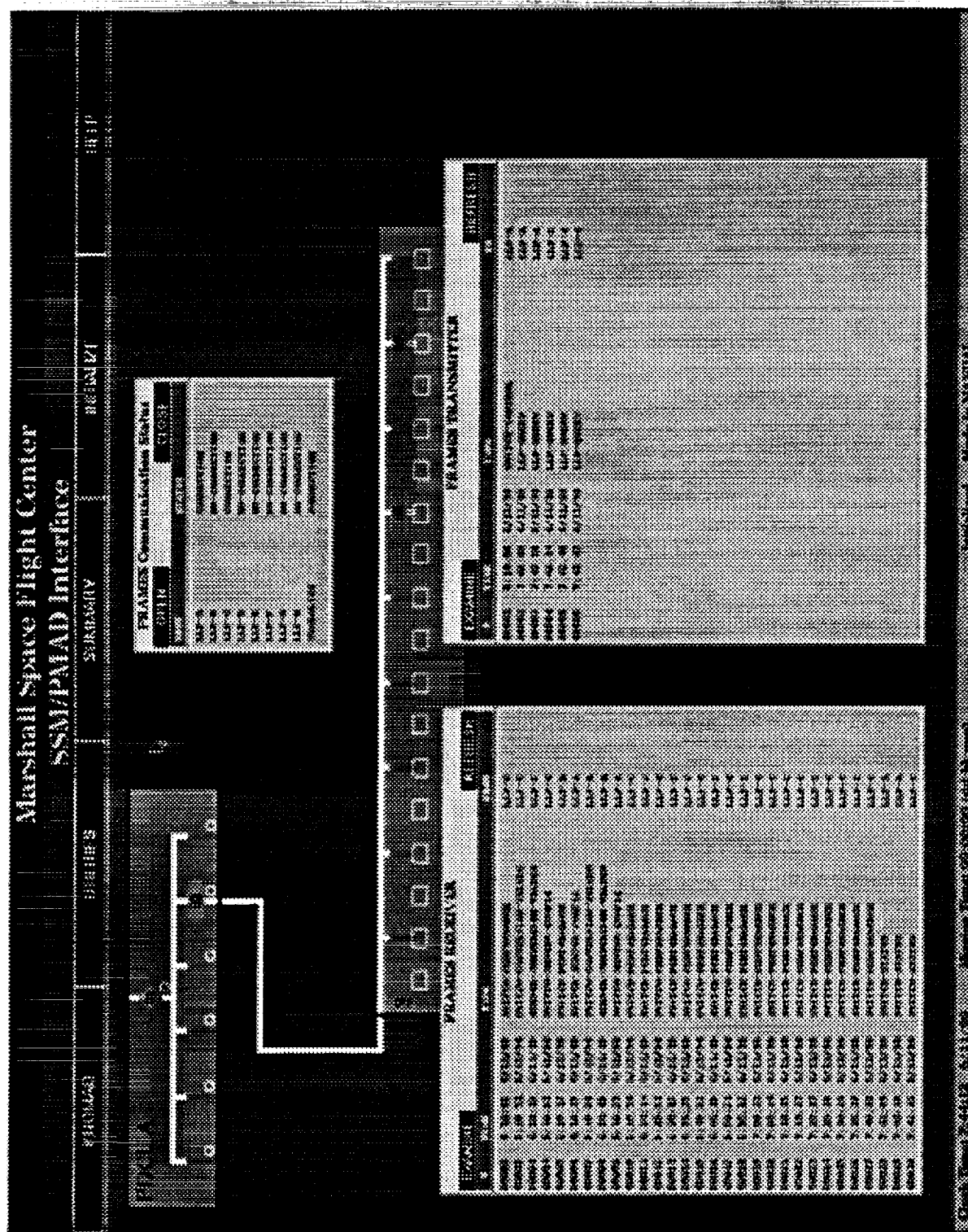
The Communications Pullright Menu The communications options menu is seen in figure 9. It is accessed by holding any mouse button down over the utilities option in the menu bar to bring up the utilities menu. The user then positions the mouse over the communications option and *pulls* the mouse to the right until the communications options sub-menu appears, all the while holding the mouse button down.

The communications menu allows the user to monitor both the transmitter and the receiver. It also allows the user to access the communications status of FRAMES with the rest of the system. Finally there is a help option, shown in figure 11.

Selecting one of the communications options—the transmitter, receiver, or status—will bring up the corresponding window on the interface. These are shown in figure 12. These windows are a specialized version of pop-up display windows. The user may update their contents, move them and hide them as with ordinary pop-up display windows. However, these windows also have additional options.

The transmitter and receiver windows allow the user to examine any of the transmitted messages between FRAMES and the rest of the SSM/PMAD breadboard. This is not at all restrictive since the SSM/PMAD interface is the central part of the breadboard. The user may examine an entry by clicking any mouse button while it is positioned over the examine

⁴It is assumed in this manual that the user has some knowledge of both compressed files and raster files.



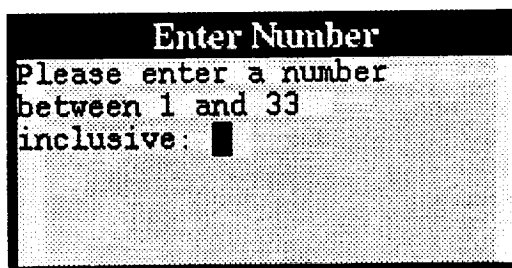


Figure 13: A Prompt Window for Displaying a Transaction

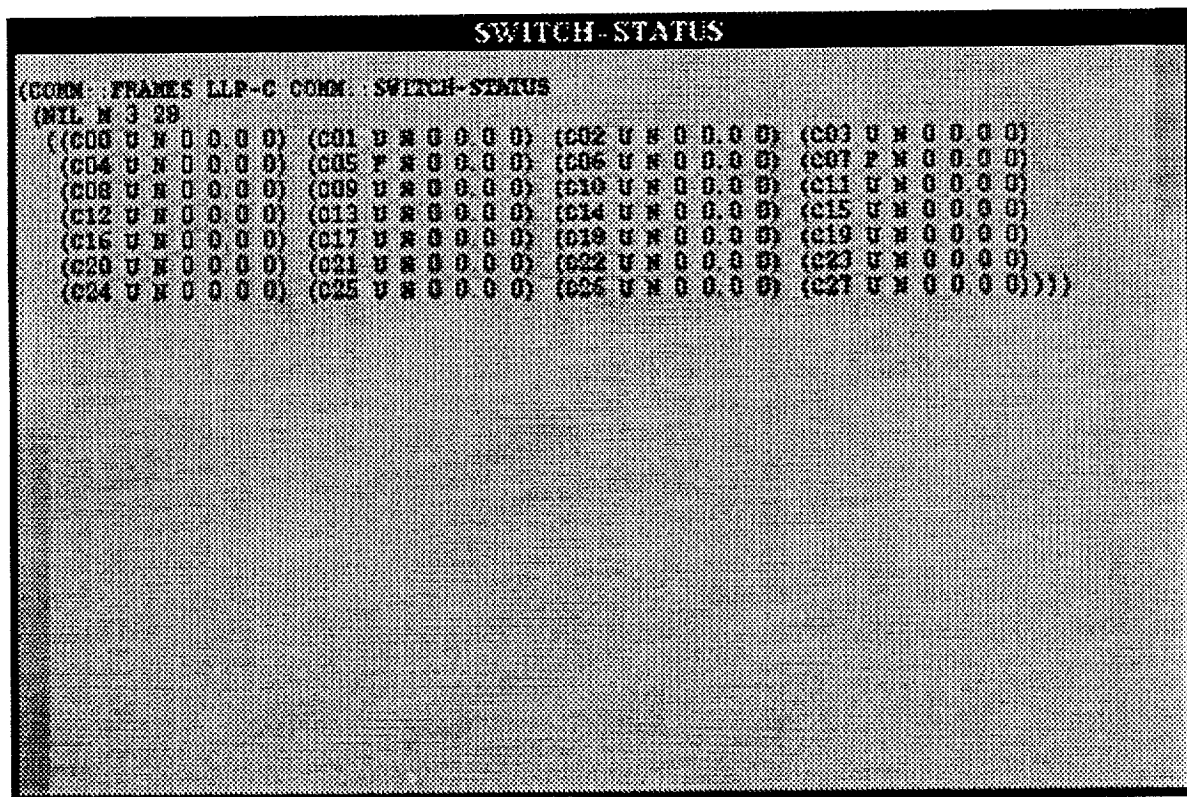


Figure 14: A FRAMES Transaction

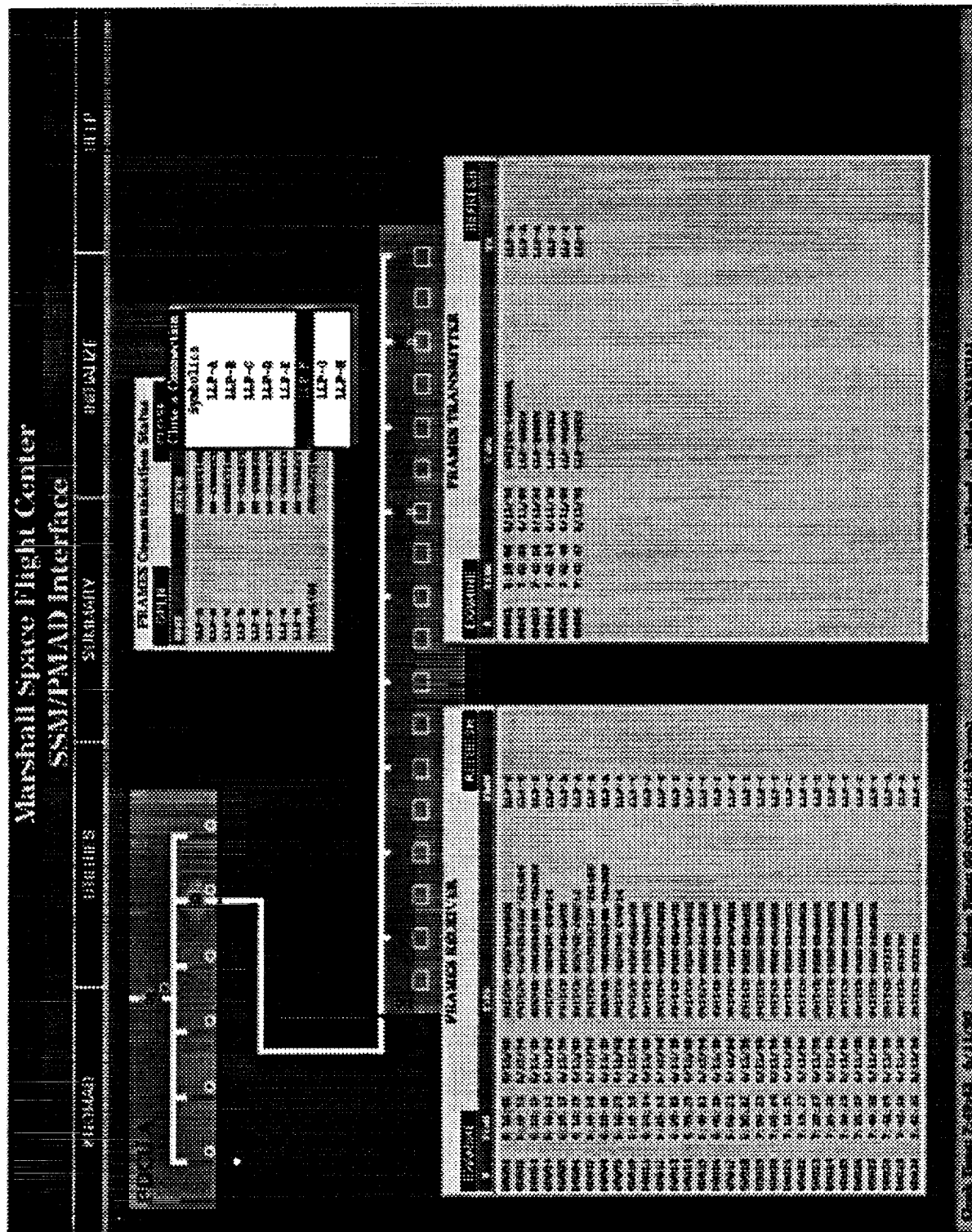


Figure 15: Closing a Communications Connection

entry option on the window. This will bring up a window as shown in figure 13. The user then types the number of the transaction to be displayed while the mouse is positioned over that window. When the user types a return to input the number the transaction will be displayed in a pop-up display window. An example is shown in figure 14.

The user may monitor and manipulate communications connections between FRAMES and the rest of the breadboard. If the user holds down any mouse button over either the open or close option of the communications status window a pop-up menu will be displayed for the user to select which other computer the user wants to open or close a connection with. An example is shown in figure 15. From the figure it is easy to see that the user may close a connection to any of the other computers. However, when opening a connection the user will only be able to open a connection to the Symbolics computer. This is because the LLPs are responsible for opening and maintaining their own connections.

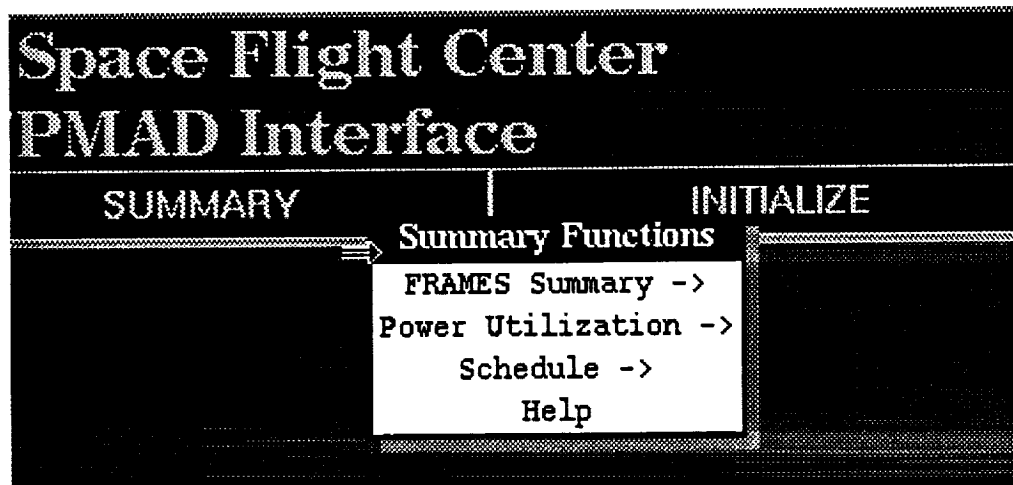


Figure 16: The Summary Menu

The Summary Menu The summary menu options are shown in figure 16. The user has access to three sub-menus through the summary menu. These are a switchgear summary menu, a power utilization menu, and a schedule menu and are discussed in the following paragraphs. The summary help window is shown in figure 20.

The Switchgear Summary Pullright Menu The FRAMES summary sub-menu is shown in figure 17. This menu should really be called the switchgear summary menu. It allows the user to get a summary of switchgear components, RPCs and sensors, in a number of ways. The user can get a full summary of the system, a summary by LLP, or a summary by selected components. These options will bring up display windows that display the data on switches and sensors in an easy to read format. However, these functions are not currently

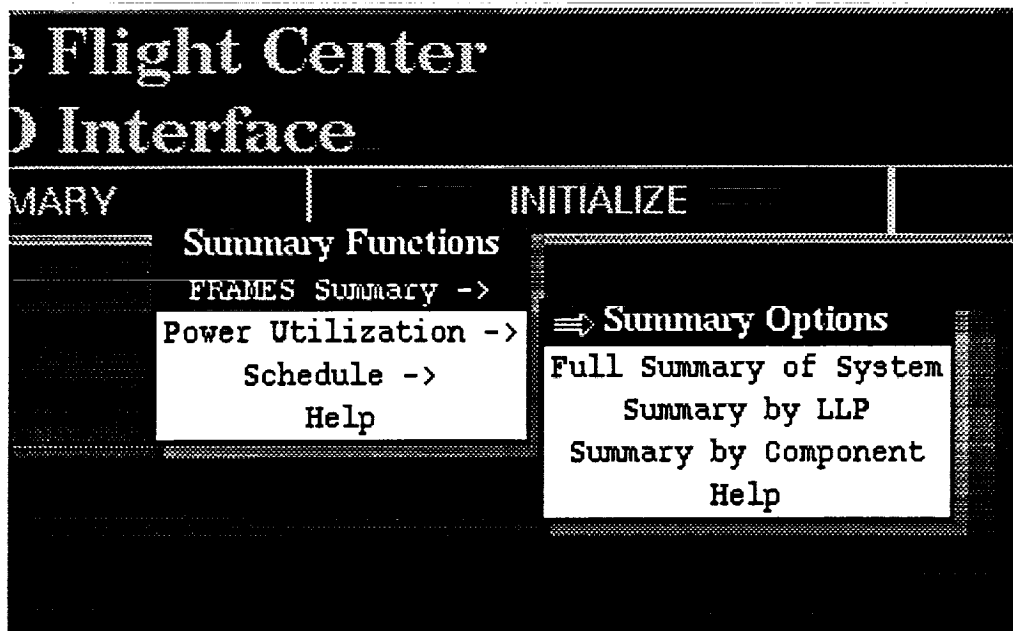


Figure 17: The Switchgear Summary Menu

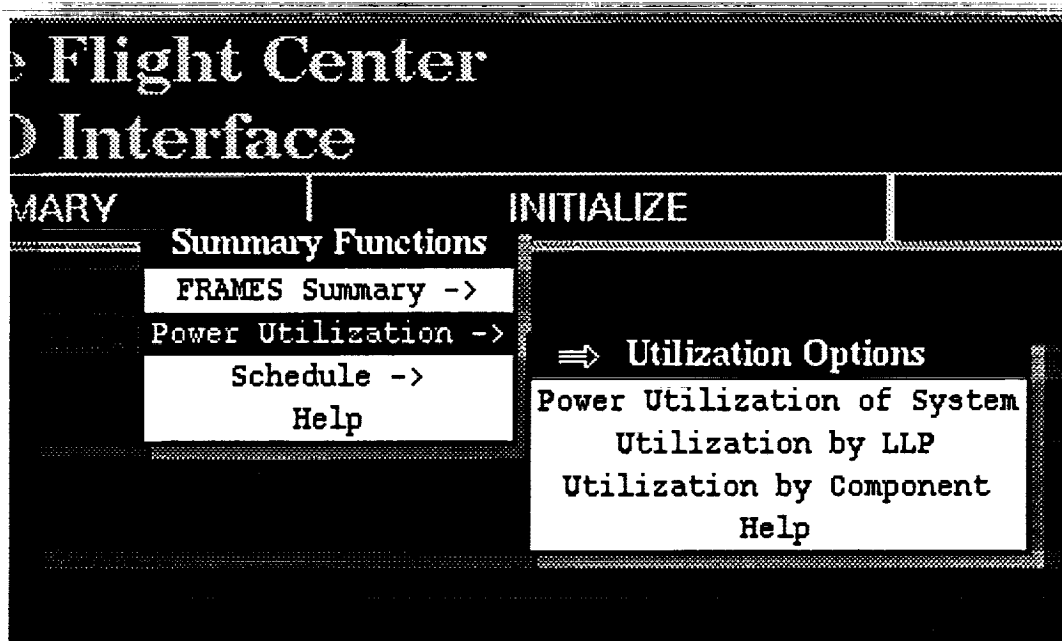


Figure 18: The Power Utilization Options Menu

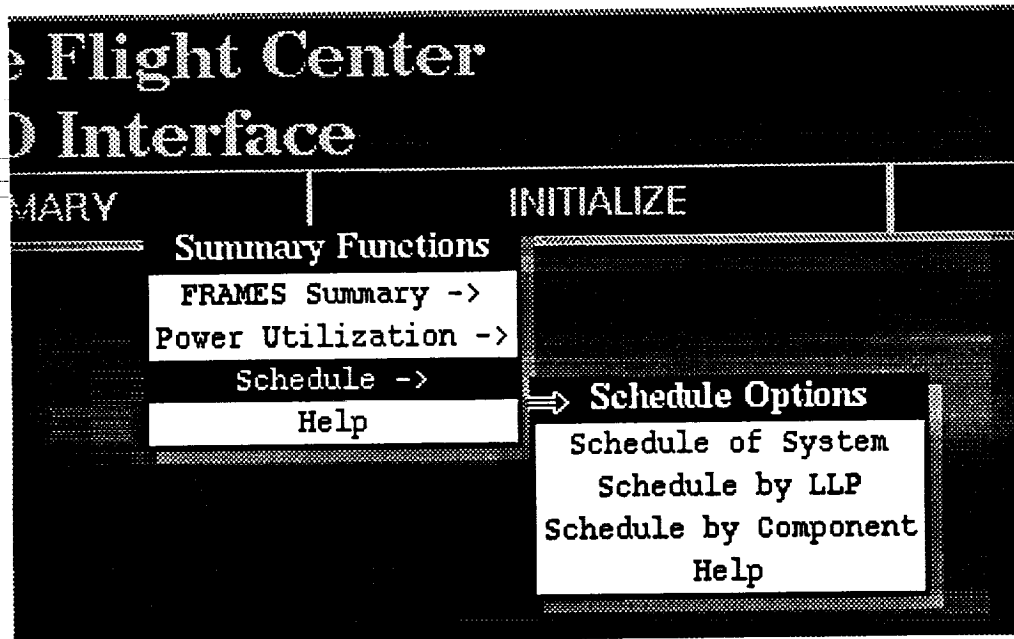


Figure 19: The Schedule Options Menu

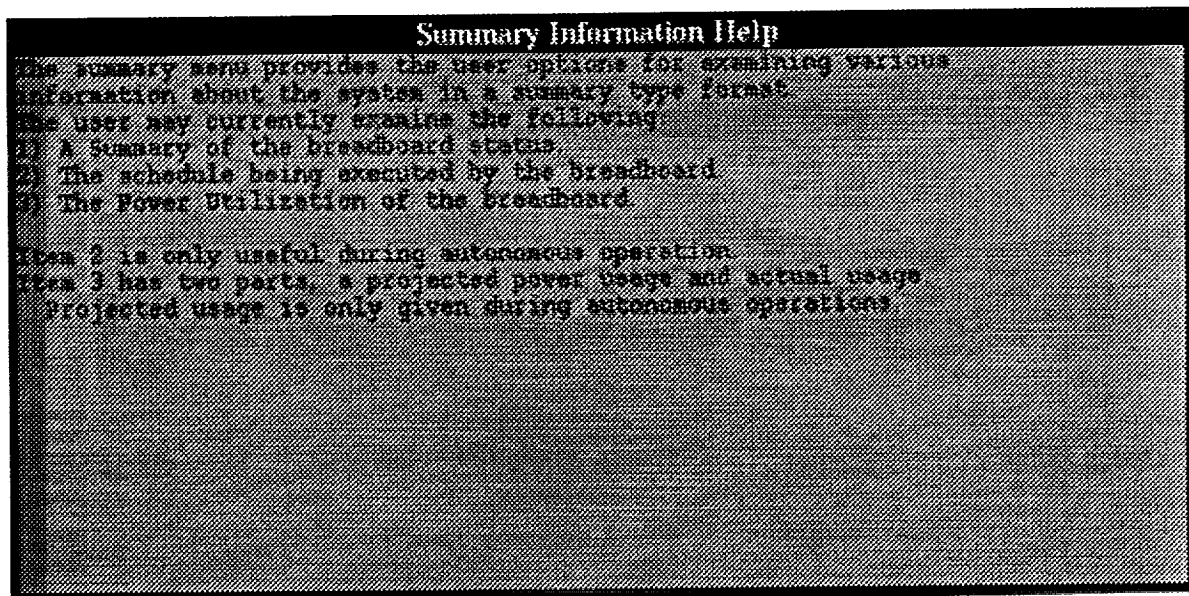


Figure 20: The Summary Help Window

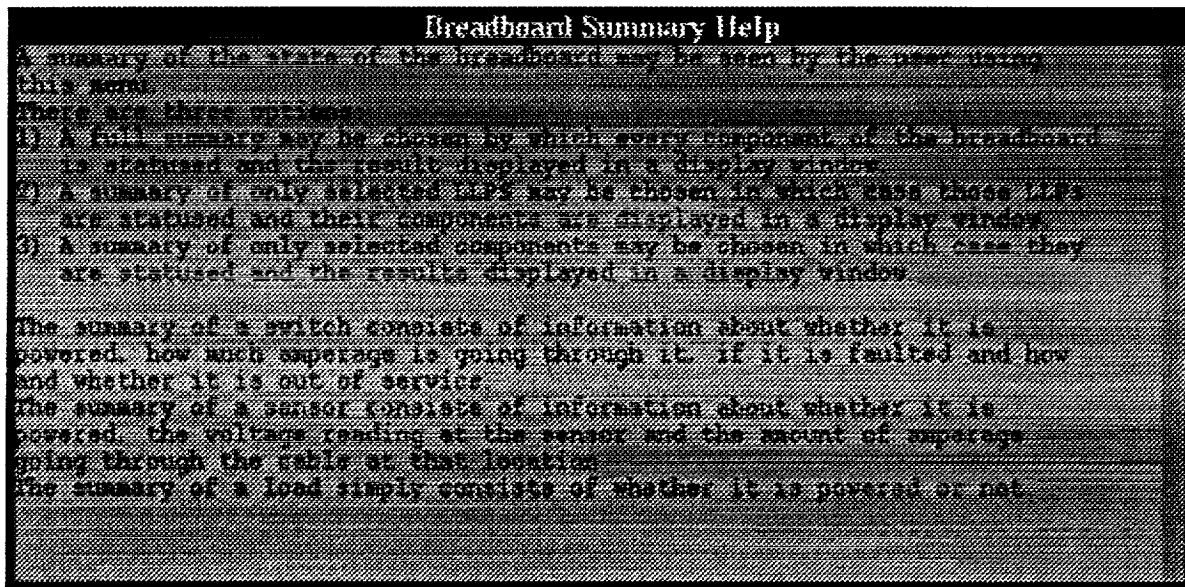


Figure 21: The Switchgear Summary Help Window

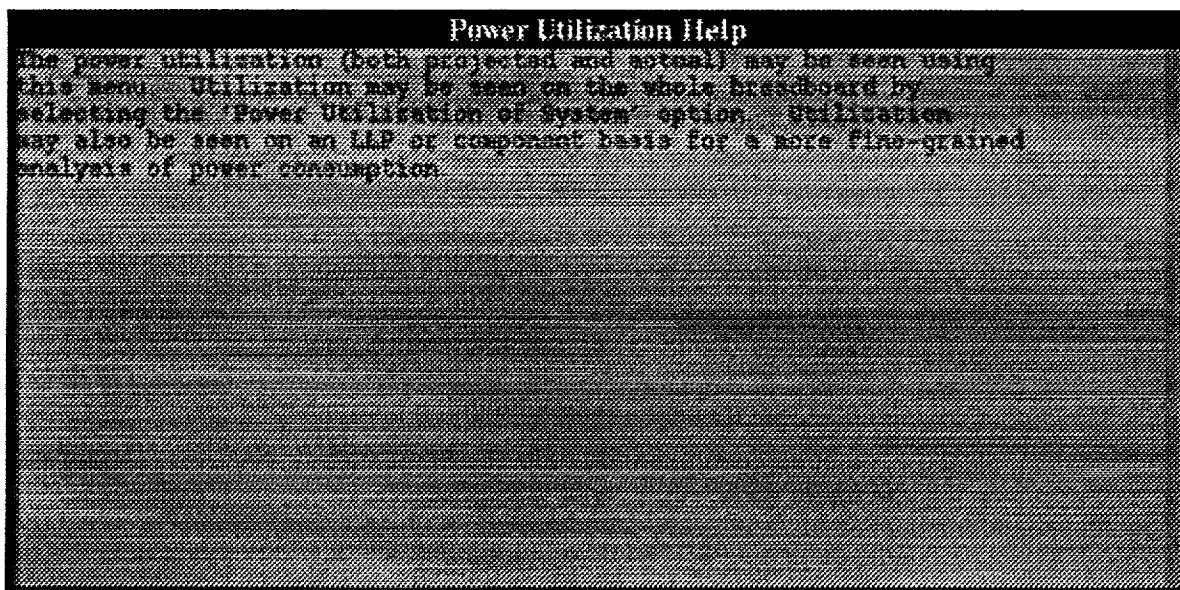


Figure 22: The Power Utilization Help Window

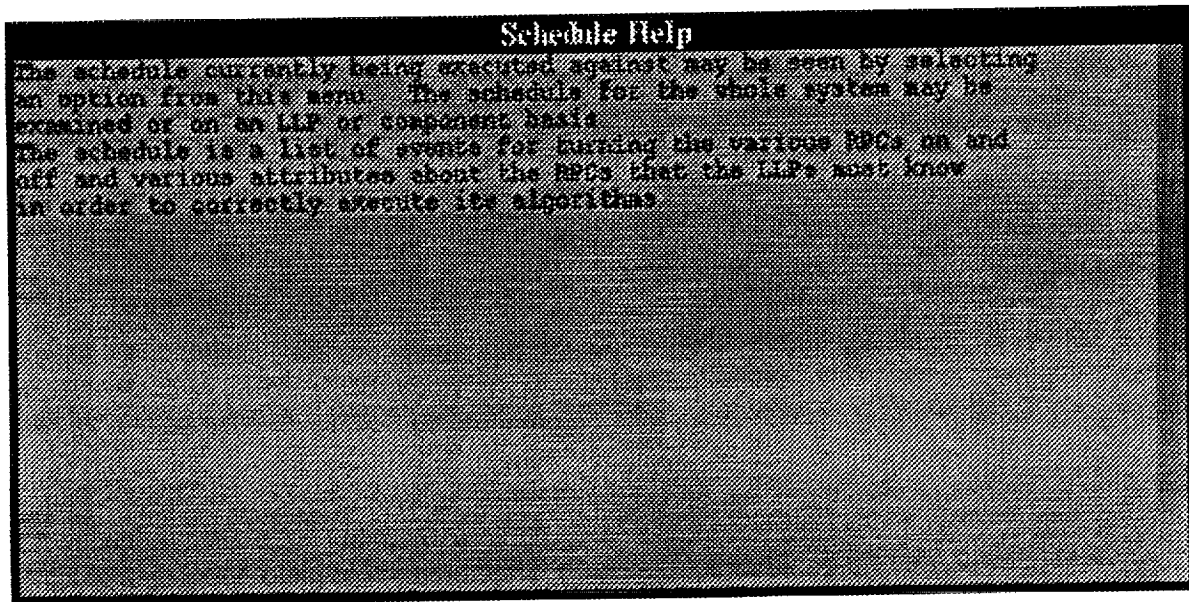


Figure 23: The Schedule Help Window

implemented. Currently, the user will need to get data on a switch or a sensor by accessing the component directly.

The switchgear summary help menu is shown in figure 21.

The Power Utilization Pullright Menu The power utilization sub-menu is shown in figure 18. The user has access to graphs that display power usage data, both scheduled and actual, by selecting these options. The user may choose to see the utilization of the entire breadboard, or just an LLP, or of selected components. However, these functions are currently not implemented.

The power utilization help display window is shown in figure 22.

The Schedule Pullright Menu The schedule sub-menu is shown in figure 19. The user may access the schedule for the entire breadboard, for an LLP, or for selected RPCs. The schedule will be displayed that shows the user when the various RPCs will be turned on and off as well as other information related to the scheduled events. However, these functions are not currently implemented.

The schedule help display window is shown in figure 23.

The Initialize Menu The first initialization menu the user will see is shown in figure 24. When the SSM/PMAD breadboard is not initialized the user will use this menu to initialize it. It has two options, the initialize FRAMES option which is a pull-right to a sub-menu of initialization options, and a help option. The help display window is shown in figure 26.

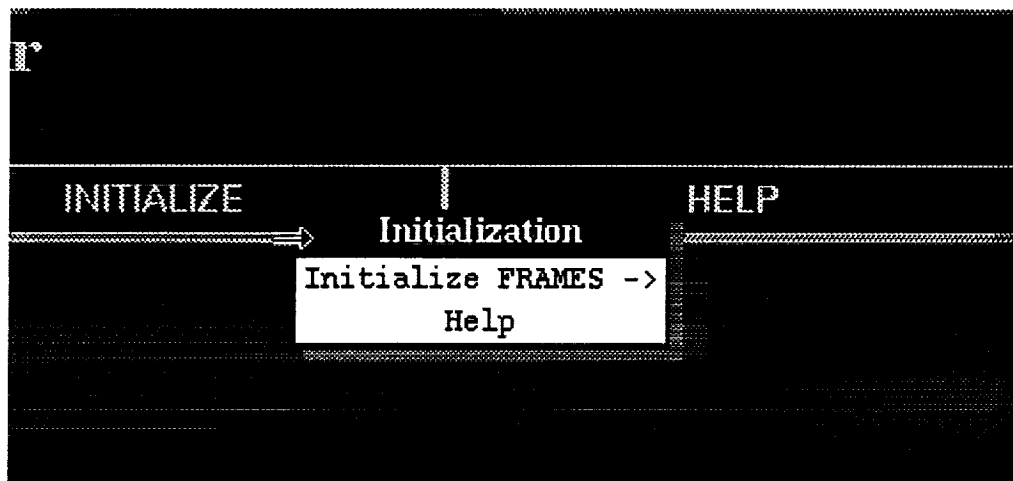


Figure 24: The Initialize Menu

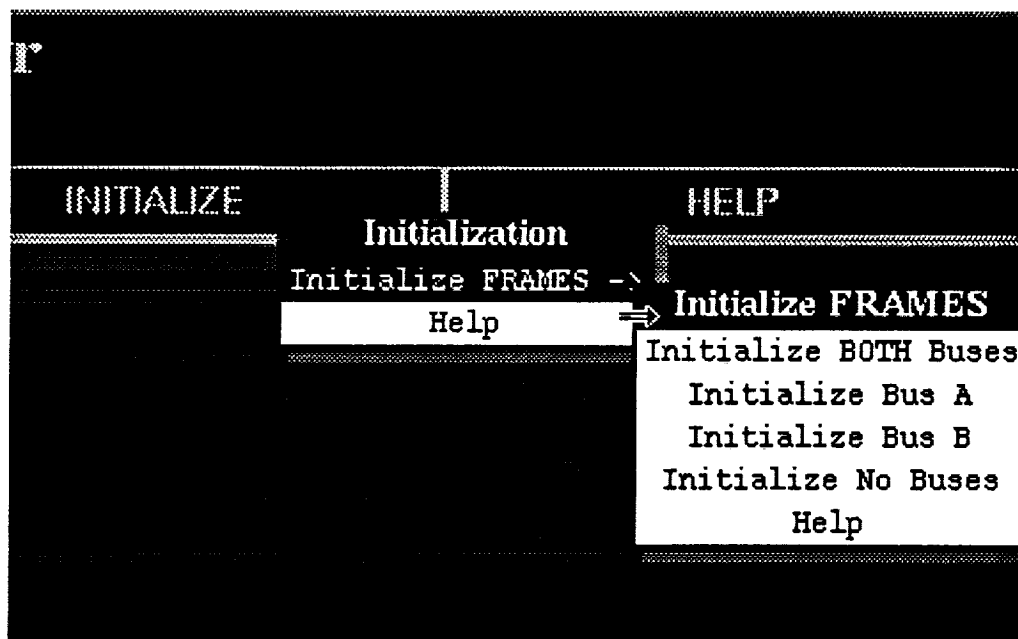


Figure 25: The Initialization Options Menu

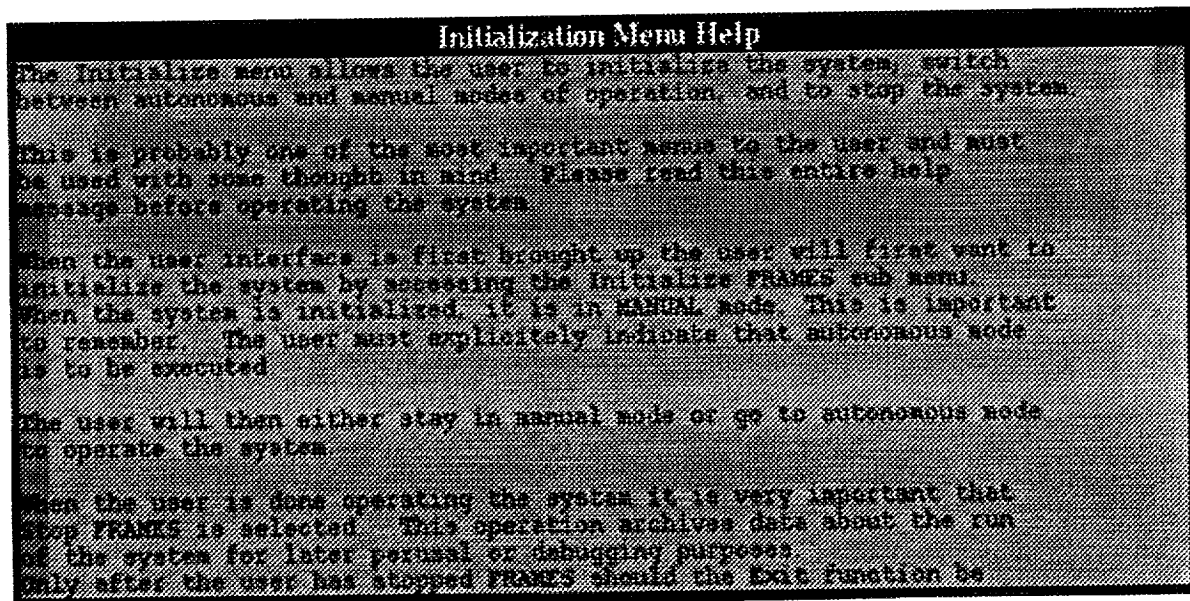


Figure 26: The Initialize Help Window

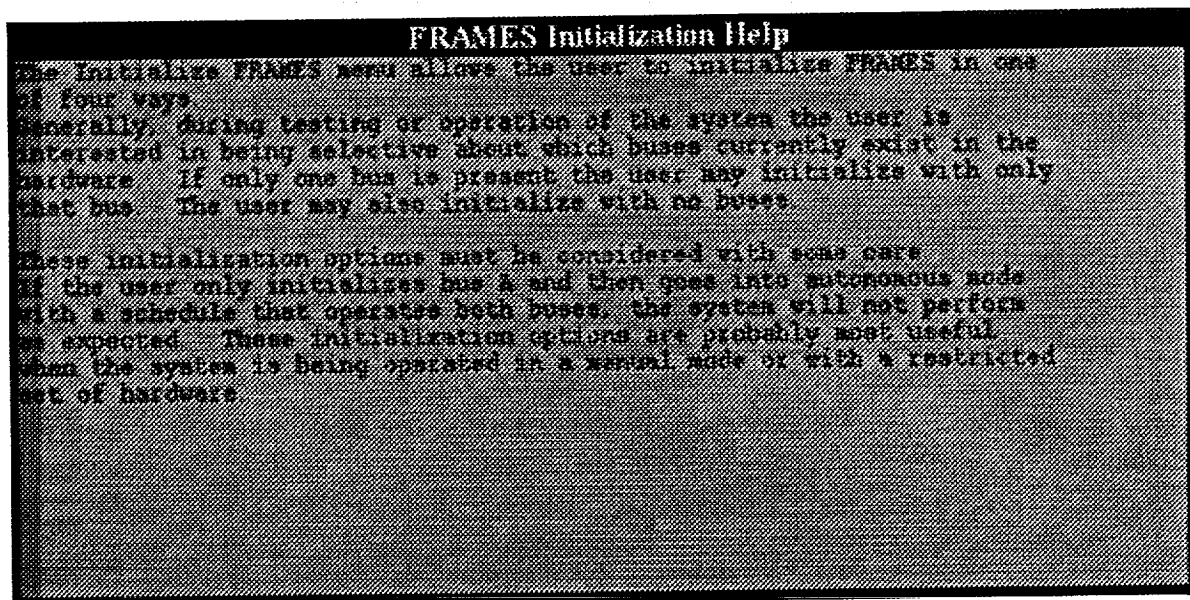


Figure 27: The FRAMES Initialization Help Window

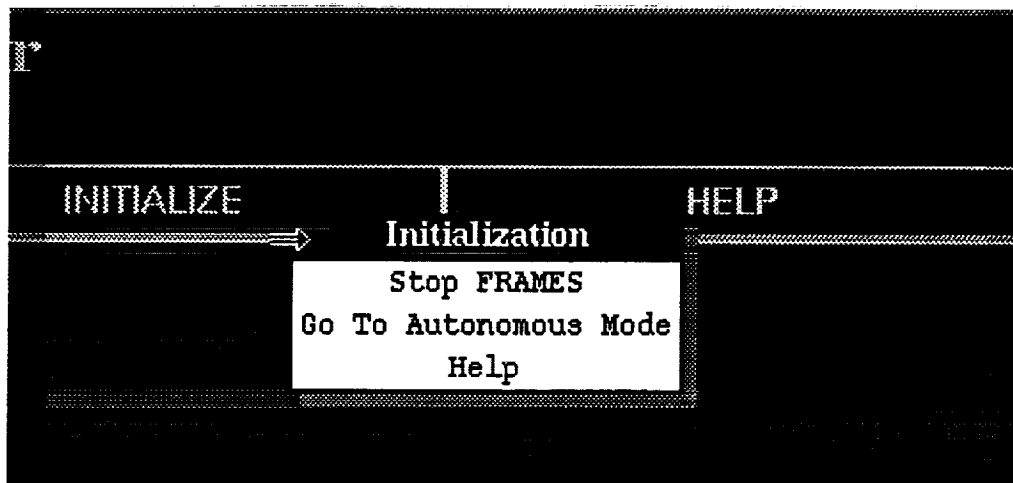


Figure 28: The Other Initialize Menu

To initialize the breadboard the user will use the initialize FRAMES option and access the pull-right sub-menu for a particular initialization choice. This is discussed in the next paragraph.

The Initialize FRAMES Pullright Menu The initialize FRAMES sub-menu is shown in figure 25. This menu allows the user to initialize FRAMES in four different configurations depending upon how power is set on the two buses to the power system. The help display window is shown in figure 27.

If power is only available on bus A, then the initialize bus A option should be chosen. The other options are chosen similarly. These options will result in the switches for the PDCU of that bus to be turned on allowing power to the LCs. There is a limitation in this version of the SSM/PMAD breadboard that requires the breadboard to be operated in this fashion (see the redundancy bug in the known bugs appendix of this use manual).

The Initialize Menu 2 When the SSM/PMAD breadboard has been initialized, the initialize menu option changes to display the menu shown in figure 28. This menu allows the user to stop FRAMES and toggle between autonomous and manual modes. Initially the user is started in manual mode and the option to go to autonomous mode is enabled. When the user is in autonomous mode the option changes to go to manual mode.

The stop FRAMES option is used to shut down the SSM/PMAD breadboard. This function archives data for debugging purposes. It also allows the user to exit the SSM/PMAD interface.

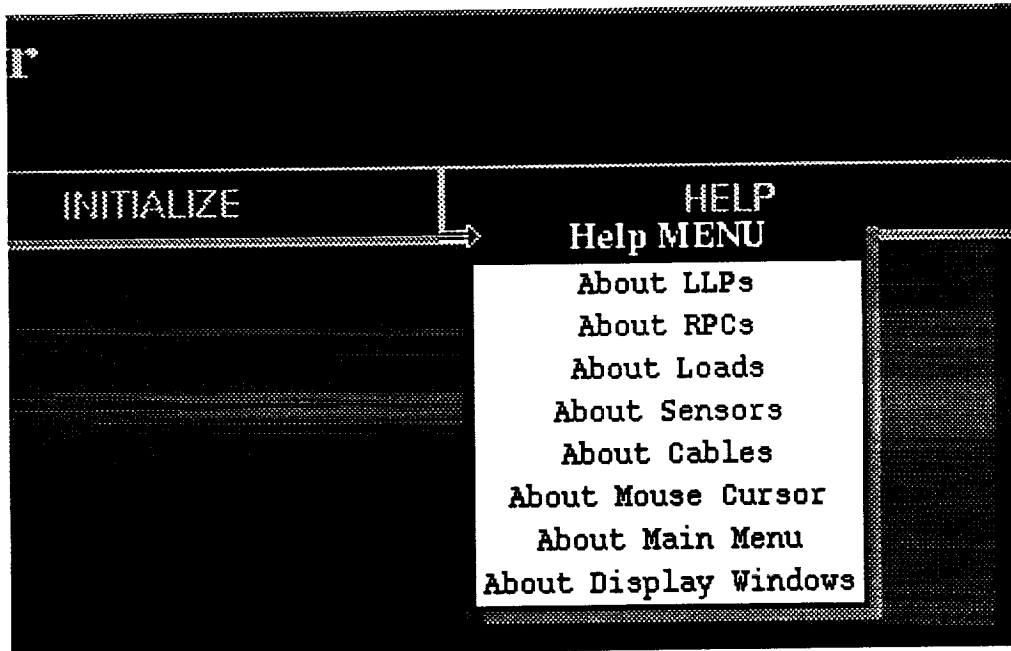


Figure 29: The Help Menu

The Help Menu The help menu is shown in figure 29. It allows the user to display help about various parts of the SSM/PMAD interface.

The about LLPs option allows the user to access information that describes the LLPs and options available to the user from the LLPs. This display window is shown in figure 30. These functions are discussed in section 4.3.3.

The about RPCs option allows the user to access information that describes the RPCs and options available to the user from them. This display window is shown in figure 31. The functions associated with the RPCs are discussed in section 4.3.3.

The about loads option allows the user to access information that describes the loads and options available to the user from them. This display window is shown in figure 32. The functions available from the loads are discussed in section 4.3.3.

The about sensors option allows the user to access information that describes the sensors and options available to the user from them. This display window is shown in figure 33. The functions accessible from the sensors are discussed in section 4.3.3.

The about cables option allows the user to access information that describes the cables and options available to the user from them. This display window is shown in figure 34. The functions accessible from the cables are discussed in section 4.3.3.

The about mouse cursor help option brings up a display window describing the meanings of the various mouse cursors used on the SSM/PMAD interface.

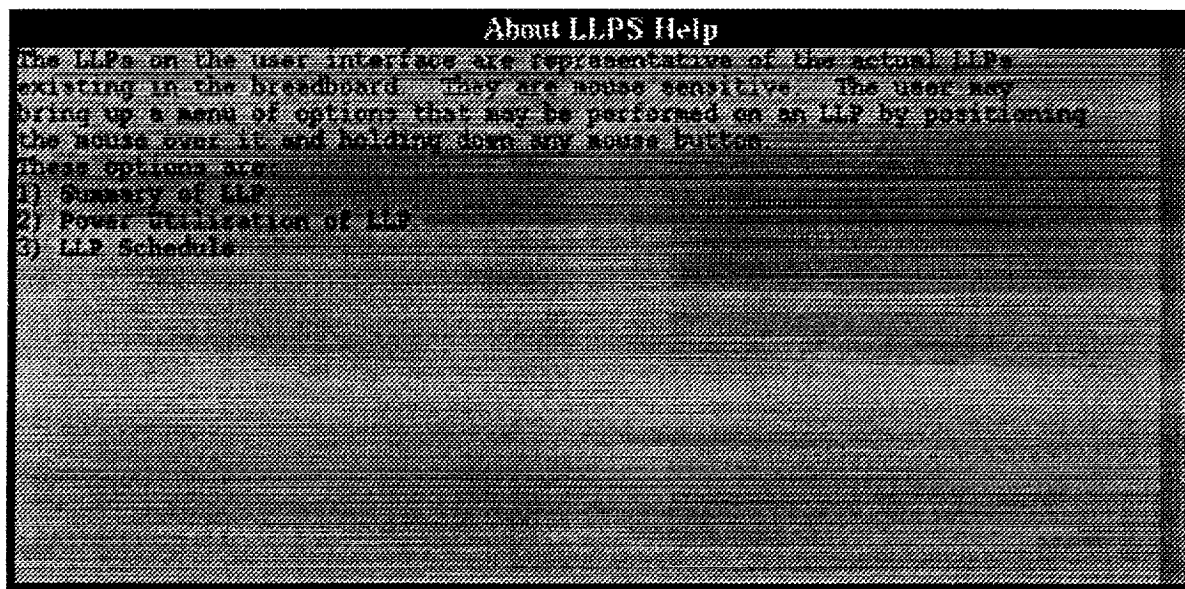


Figure 30: The LLPS Help Window

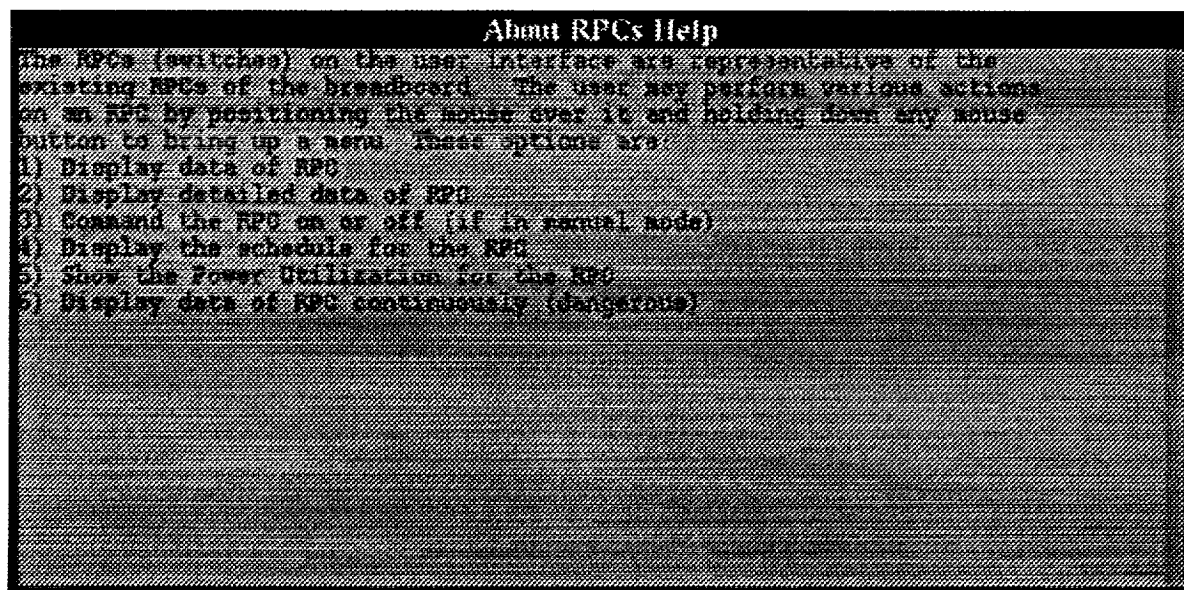


Figure 31: The RPCs Help Window

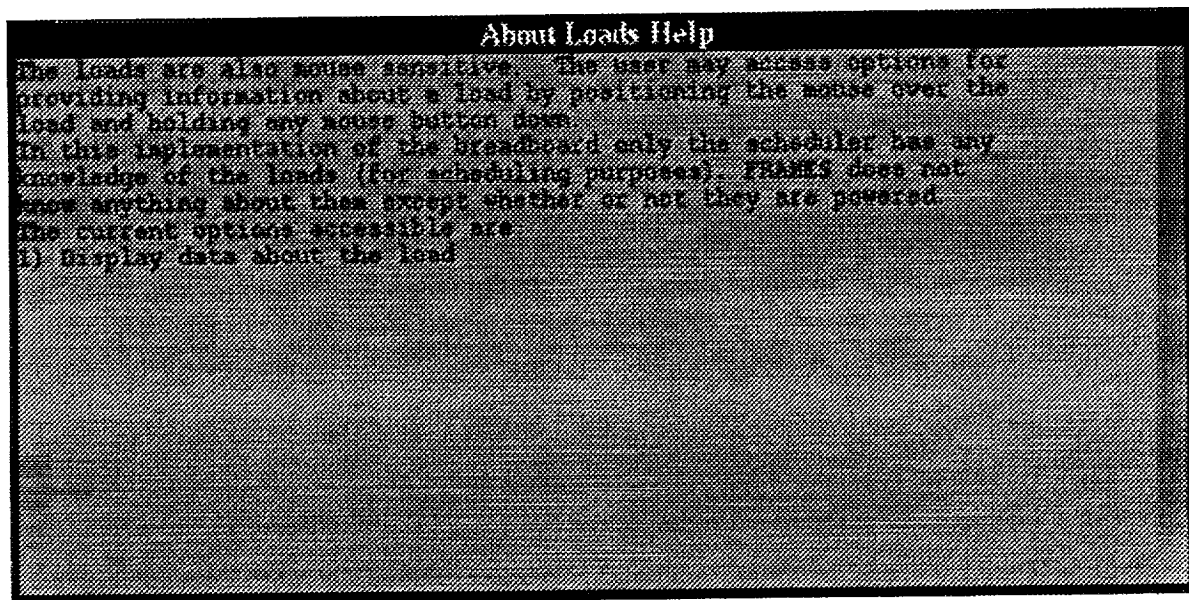


Figure 32: The Loads Help Window

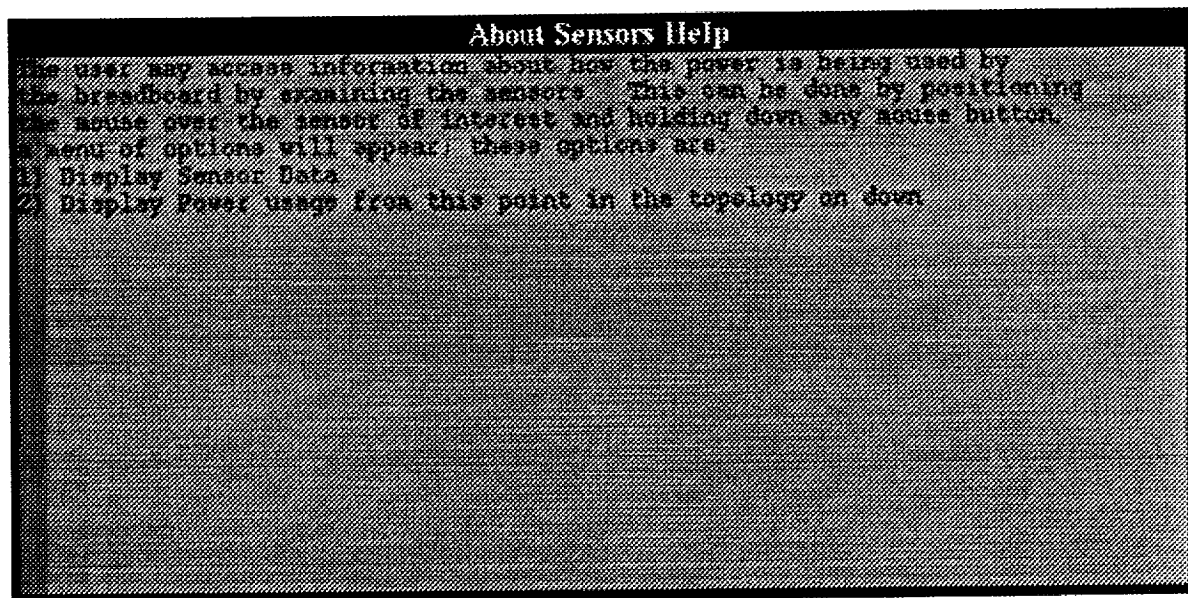


Figure 33: The Sensors Help Window

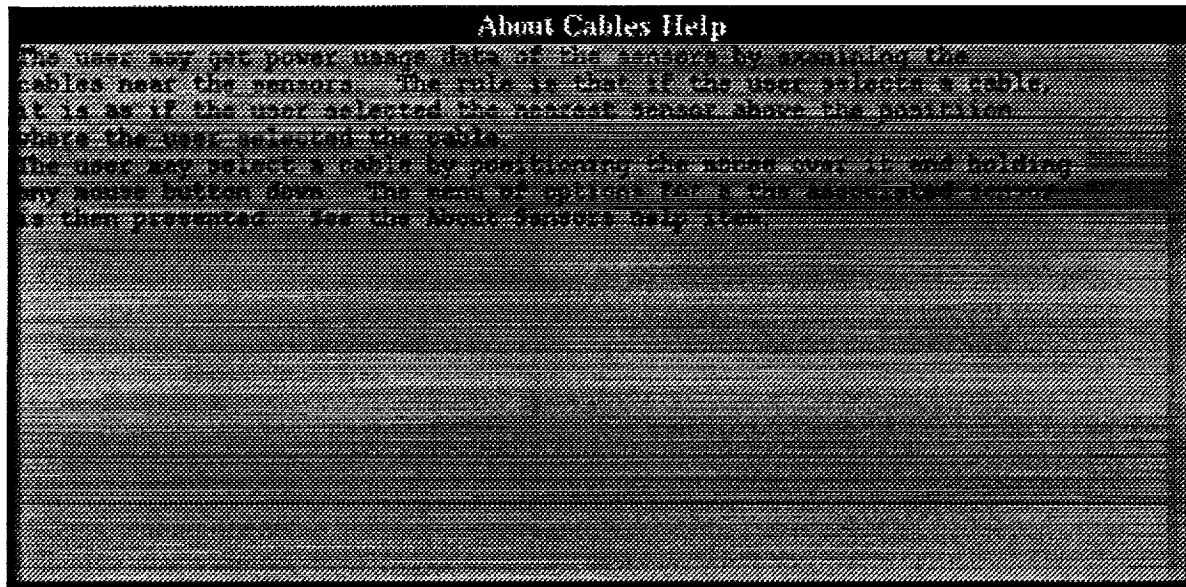


Figure 34: The Cables Help Window

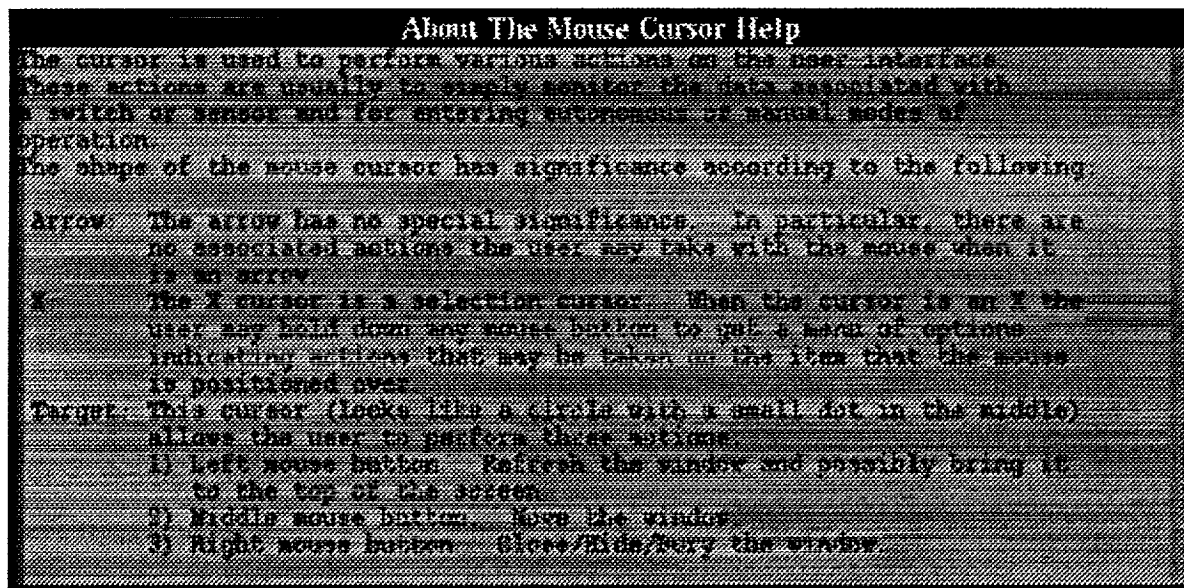


Figure 35: The Mouse Cursor Help Window

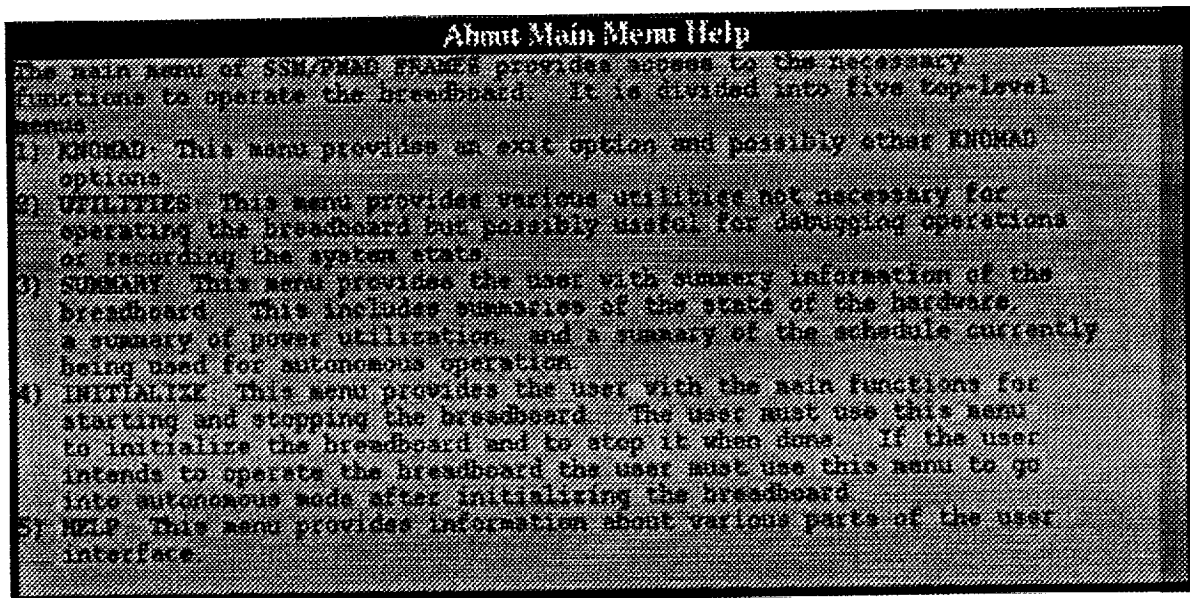


Figure 36: The Main Menu Help Window

The arrow mouse cursor is used to signify a normal mouse with no special semantics attached to it. The X and dot mouse cursors are both used to indicate that there is a user interface menu option available. While the mouse is an X or a dot, the user may hold down any mouse button and a menu will appear that is context sensitive to the item the mouse is positioned over. The target cursor is used when the mouse is positioned in the title bar of a pop-up display window. Each mouse button has a different semantics when pressed while in the title bar. The left button is used to bring a partially exposed window to the top as well as update any data in it, the middle button is used to move the window and the right button is used to hide the window.

The about main menu option brings up a display window describing the main menu bar of the interface. These functions are the subject of this entire section of the document.

The about display windows option brings up a pop-up display window that describes the actions a user has available on pop-up display windows. The attributes of a pop-up display window have been described earlier and throughout this document where it has been relevant.

4.3.3 The Power System Components' Functions

The various components of the power system representation on the SSM/PMAD interface that many be manipulate are the LLPs, RPCs, sensors, cables, and loads. An example screen with these items on it is shown back in figure 3. Each of these items may be moused by the user to bring up a menu of options the user has on the component clicked on and are

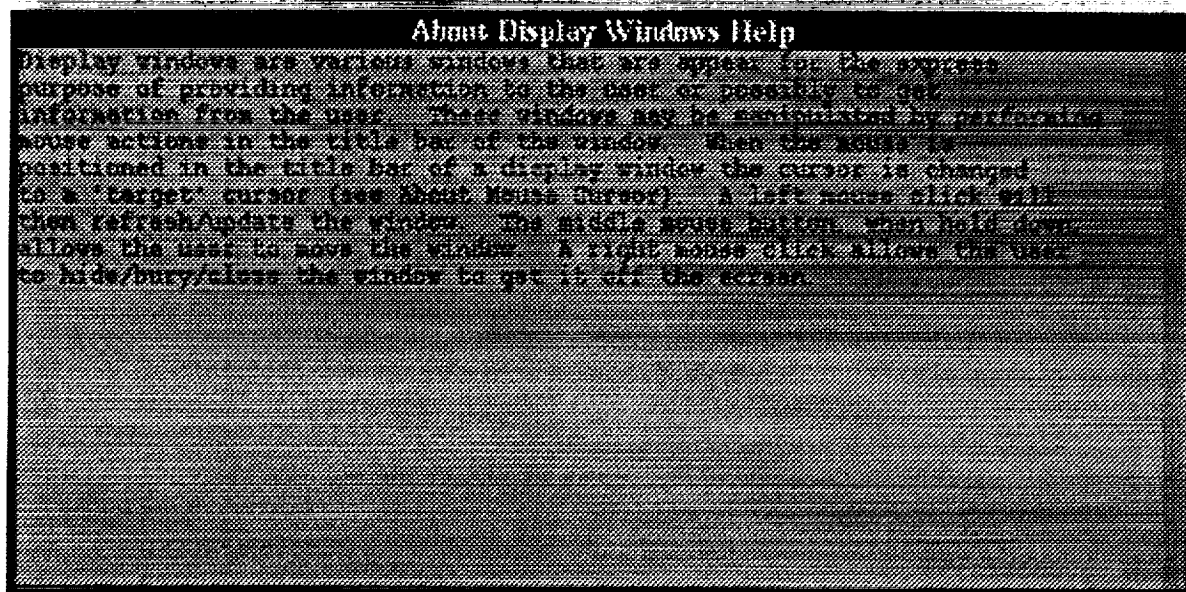


Figure 37: The Display Windows Help Window

discussed in the following paragraphs.

The LLPs The LLPs on the interface represent the LLPs of the SSM/PMAD breadboard. When the mouse is positioned over an LLP the mouse cursor changes to a dot. This indicates that if any mouse button is pressed down, a menu of options will pop-up for the user to select about an LLP. This menu is shown in fig 38.

The user may examine the schedule, the power utilization, or a summary of the LLP by selecting one of the options. These functions are the same functions as those available in the summary menu. However, these are not implemented in version 1.0 of the SSM/PMAD interface.

The RPCs When the mouse cursor is positioned over an RPC the mouse cursor changes to an X cursor. If the user holds down any mouse button while positioned over an RPC a menu of RPC options will be displayed. This is shown in figure 39.

The user may select to get normal or detailed data from an RPC as well as continuous (normal) data from it. Each of these functions will bring up a pop-up display window with the selected RPC as the title. This is a special version of a pop-up display window and is not scrollable. The usual title bar functions are available: left-click to get new/updated data, middle-press to move the window, and right-click to close the window.

A normal switch data window will display information about how much current is being used by the switch, the switch's state, if it is powered or not, how it has been tripped (if it has been tripped), and if the switch is available. Continuous switch data uses exactly

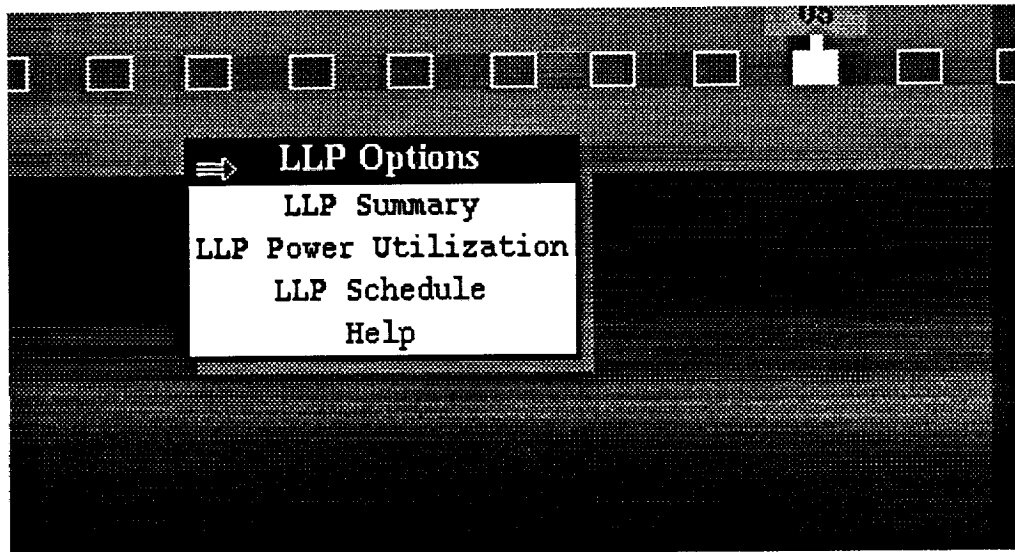


Figure 38: An LLP Menu

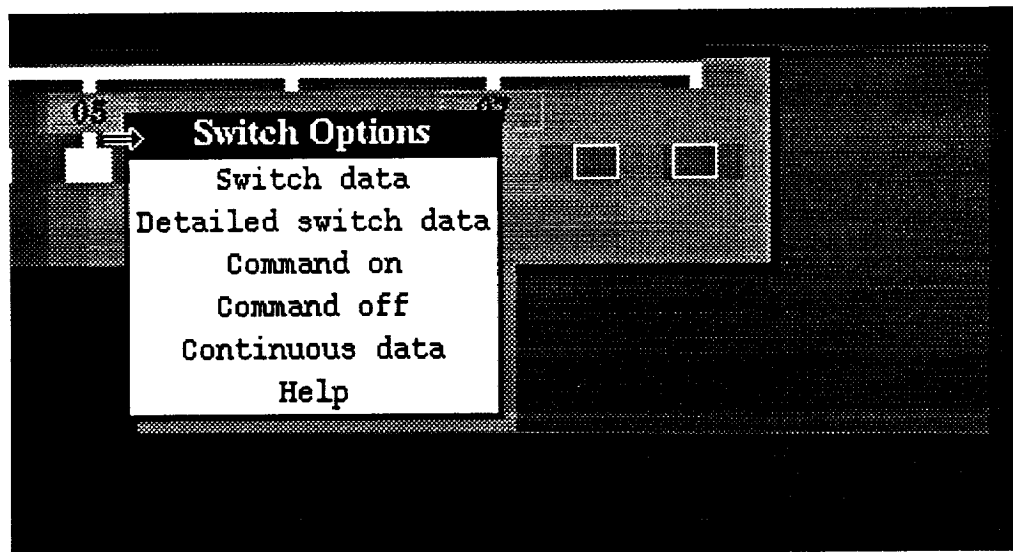


Figure 39: An RPC Menu

the same window with the same data but is updated continuously with data that the LLP gathers from the RPC every time through its loop. Detailed switch data provides advanced information about the switch from the LLP. Figure 43 shows an example of normal switch data.

In addition to these functions the user may also command a switch on or off if the user is in manual mode. These options do not appear if the user is in autonomous mode. If the user selects one of these options, the corresponding command is sent to the LLP and the interface is updated appropriately. An interesting feature of the command options is that the user does not need to manually turn on each switch in the power path to enable a lower switch to be turned on. For example, suppose no power is on anywhere in the breadboard. The user then wishes to command on switch C05. The user may directly access the command on function of RPC C05 and command it on. The function will then realize that neither A01 or A03 are on and turn them on first to enable switch C05. This type of interaction with the breadboard will also occur when a switch is commanded off. Any lower switches will be turned off first.

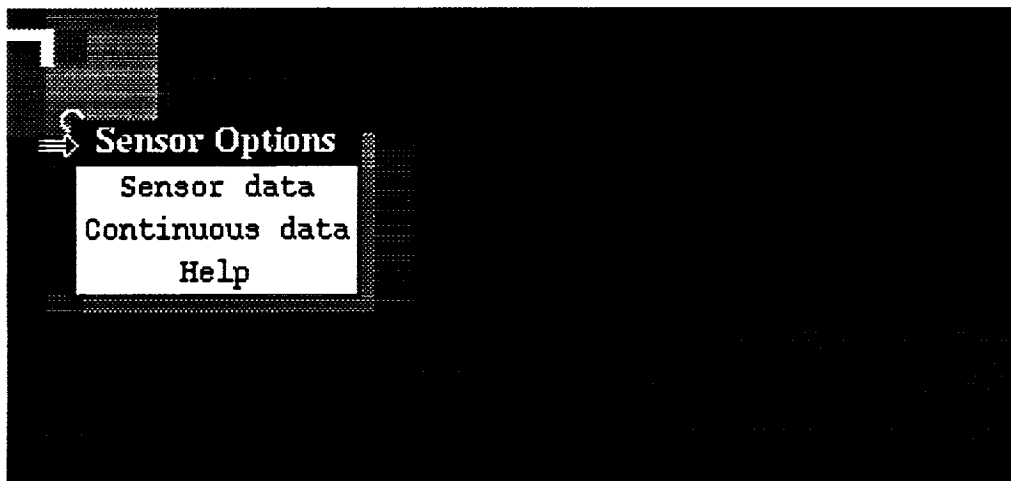


Figure 40: A Sensor Menu

The Sensors When the mouse cursor is positioned over a sensor the mouse cursor changes to an X cursor. If the user holds down any mouse button while positioned over a sensor a menu of sensor options will be displayed. This is shown in figure 40.

The user may choose to access sensor data about the sensor as the only option available in version 1.0 of the SSM/PMAD Interface. This option, like the switch data option of RPCs, will pop-up a specialized pop-up display window to display current, voltage, power and temperature of the sensor. An example is shown in figure 44.

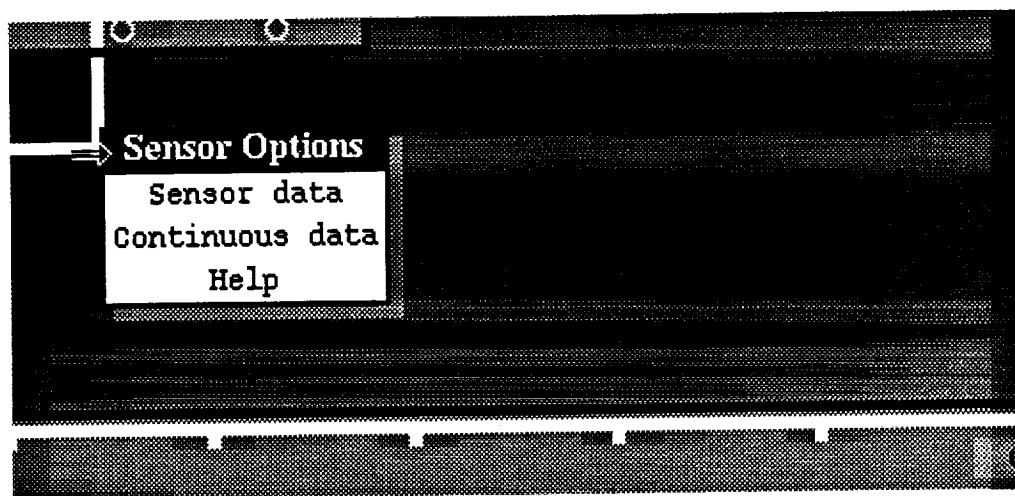


Figure 41: A Cable Menu

The Cables When the mouse cursor is positioned over a sensor the mouse cursor changes to an X cursor. If the user holds down any mouse button while positioned over a sensor a menu of sensor options will be displayed.

When the user chooses to get information about a cable the system automatically directs that action to a sensor button action. The reasoning is that a cable, by definition, cannot be accessed and cannot provide information about itself. The sensor, however, is used to provide this information. Therefore, when the user clicks on a cable, the closest sensor above the section of the cable where the user clicked is used instead. An example of this is shown in figure 41.

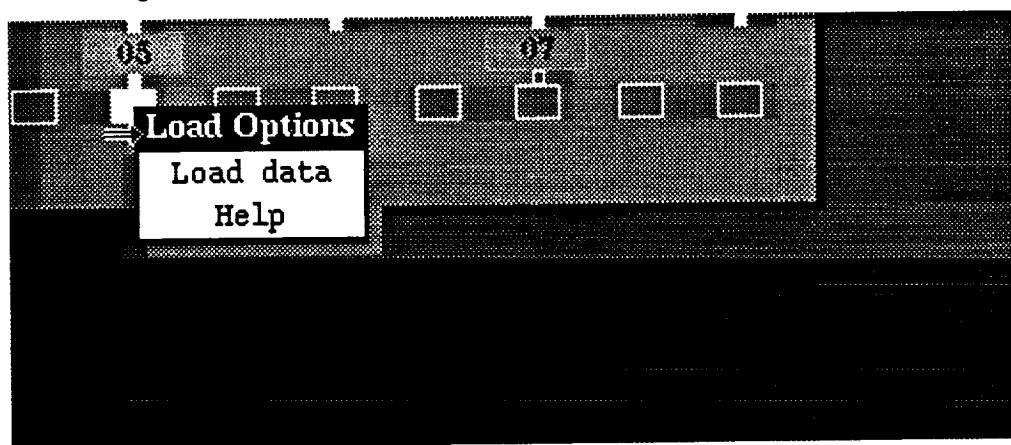
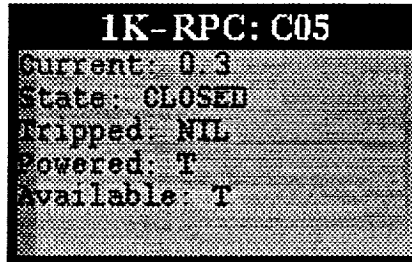


Figure 42: A Load Menu

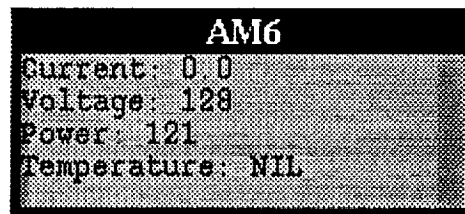
The Loads When the mouse cursor is positioned over a load the mouse cursor changes to an X cursor. If the user holds down any mouse button while positioned over a load a menu



A screenshot of a terminal window displaying switch data. The title bar reads "1K-RPC: C05". The data is as follows:

Parameter	Value
Current	0.3
State	CLOSED
Tripped	NIL
Powered	T
Available	T

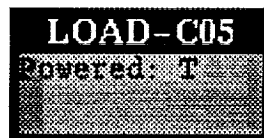
Figure 43: Switch Data



A screenshot of a terminal window displaying sensor data. The title bar reads "AM6". The data is as follows:

Parameter	Value
Current	0.0
Voltage	128
Power	121
Temperature	NIL

Figure 44: Sensor Data



A screenshot of a terminal window displaying load data. The title bar reads "LOAD-C05". The data is as follows:

Parameter	Value
Powered	T

Figure 45: Load Data

of sensor options will be displayed. This is shown in figure 42.

The options available to the user for a load are only to get data about the load. In version 1.0 of the SSM/PMAD Interface, no knowledge of the loads is maintained. Therefore, only data indicating whether a load is powered or not is given (shown in figure 45). This also explains why, although only those switches that are available in the power system switchgear, all the loads are always displayed. No knowledge of whether there are even any loads in existence is even known, therefore, all the loads are simply displayed whenever an LLP is displayed.

A Known Bugs

This section provides a brief description of important problems with the SSM/PMAD breadboard that are known at this printing.

1. Scrolling problems. The display windows scroll fairly well but occasionally exhibit anomalous behavior. The scrolling for the FRAMES receiver and transmitter logs barely works. The transactions logs will not print some lines where they should be. When there are more entries than fit in the window, the numbering is off by one. If an entry is labeled 52, the number is really 53 if it is to be examined.
2. Examining a transaction entry has an interesting bug. When the user types in a number, it is echoed twice. Just ignore the extra echo.
3. The SSM/PMAD breadboard does not handle redundant switching completely. In version 1.0 the scheduler will shut off a switch that has been switched to redundant when contingency scheduling has occurred.

The redundancy problem has generated a number of problems in the SSM/PMAD breadboard. The most apparent is that operation of the breadboard involves initially turning on all the PDCU switches so that if a load needs to switch to its redundant supply, there will be available power. This needs to be thought about in some detail in the next few months. A proper partitioning of MAESTRO, FELES, and LPLMS is one of the implications of the handling of redundancy appropriately.

Some initial discussions suggest that MAESTRO doesn't need to do much more than it currently does (except to actually keep a load on that has switched to redundant — is this an FELES problem?), that perhaps FELES should know how the power system needs to have its switches turned on according to the schedule, and finally that LPLMS should manage the whole issue of priorities so that if a load does need to switch to redundant, the proper loads, from a global point of view, will be shed.

References

- [1] W. Miller, E. Jones, B. Ashworth, J. Riedesel, C. Myers, K. Freeman, D. Steele, R. Palmer, R. Walsh, J. Gohring, D. Pottruff, J. Tietz, and D. Britt. *Space Station Automation of Common Module Power Management and Distribution*. Technical Report Contractor Report 4260, NASA, November 1989.
- [2] Joel D. Riedesel. Diagnosing multiple faults in ssm/pmad. In *Proceedings of the 25th Intersociety Energy Conversion Engineering Conference*, 1990.

- [3] Joel D. Riedesel. *Knowledge Management: An Abstraction of Knowledge Base and Database Management Systems*. Technical Report Contractor Report 4273, NASA, January 1990.
- [4] Joel D. Riedesel. A survey of fault diagnosis technology. In *Proceedings of the 24th Intersociety Energy Conversion Engineering Conference*, 1989.
- [5] Joel D. Riedesel, Chris Myers, and Barry Ashworth. Intelligent space power automation. In *Proceedings of the Fourth IEEE International Symposium on Intelligent Control*, 1989.



Contents

1.0	LLP Reference Overview	2
2.0	LLP Software	3
2.1	Switching Operations	3
2.2	Schedule Execution	3
2.3	Load Shedding	4
2.4	Fault Reporting	4
2.5	Fault Isolation	4
2.6	Redundant Switching	5
2.7	Performance Monitoring	5
2.8	Limit Checking	5
2.9	Configuration Determination	6
2.10	Multiline Interrupt Driver	6
2.11	CMC Ethernet Driver	6
3.0	LLP Hardware	8
3.1	LLP Computing Hardware Overview	8
3.1.1	LLP Motherboard Configuration	8
3.1.2	RS-422 Card Configuration	9
3.1.3	Ethernet Adapter Configuration	9
3.2	Switchgear Overview	11
3.2.1	Switchgear Interface Controller	11
3.2.2	Generic Controller	11
3.2.3	Switches	12
3.2.4	Sensors	12
4.0	Future LLP Concerns	13
4.1	Feasible LLP Solutions	13

1.0 LLP Reference Overview

This LLP Reference contains three sections with information on both the LLP software and the LLP hardware. The LLP software functionality and configuration is discussed first. This is followed by an overview of the LLP hardware configuration. The last section of this reference examines possible future directions for the LLP hardware and software development.

2.0 LLP Software

The LLP software maintains and controls low level operations through a large number of LLFs. These software functions control switching operations, schedule execution, load shedding, fault reporting, fault isolation, and redundant switching. In addition, these functions also monitor performance, maintain load limits through limit checking, and upon power up determine the LLP switchgear configuration. The LLP software makes use of several communications software driver packages. These drivers simplify communications between the LLP, the switchgear and the Solbourne workstation running FRAMES. With the communications drivers and the LLFs, the LLP software can maintain and control a load center or a power distribution control unit.

2.1 Switching Operations

The LLP software commands switches within the load center or PDCU by issuing an on or off command for the given switch as needed. The format of these commands is defined by the LLP/SIC Interface Control Document (Appendix VI). Switching operations are mainly used for schedule execution. However, they may also be used in fault isolation, redundant switching, load shedding, and manual operations commanded from the SSM/PMAD interface.

2.2 Schedule Execution

When running in autonomous mode, the LLP executes a schedule of events for commanding specific switches on or off. These events within a schedule happen at specific times during the mission. The LLP will wait for the time of the event to pass before actually processing that event. The event also contains maximum and minimum current specifications for the load on the output of the switch. This load information is used in the limit checking function. Additionally, the event contains information on whether or not the load is redundantly sourced and if there is permission to switch to redundant power in the event of a fault.

The schedule of events, known as an event list, arrives at the LLP via communications from the SSM/PMAD interface. The format of this list is defined in the LLP/FRAMES Interface Control Document (Appendix V).

2.3 Load Shedding

Load Shedding occurs in three ways. First, an immediate source power change is received at the SSM/PMAD interface and the FRAMES system generates a shed list which is passed to the LLP. Second, the LLP may shed a load if the amperage limit check for that load is out of range. Last, a redundantly sourced load moves to the alternate bus and bumps a load on that bus of lower priority. This will only happen if there is not enough power available on the redundant bus. Load shedding is an important method of controlling power usage on a given bus.

2.4 Fault Reporting

Several classes of faults may be discovered and reported at the LLP Software. The first class is hard faults; defined to be any situation which causes a switch to physically trip. An example of this would be a short on the output of the switch. The soft fault is another type of fault which is defined to be an illegal use of current in our system. At present, only limit checking looks for this type of fault. When a fault is detected at the LLP, the LLP sends an anomalous message to the SSM/PMAD interface. At this point, FRAMES will ask all relevant LLPs to send a quiescent data message when each LLP reaches a quiescent state. A quiescent state is defined as when all the fault data is identical to the fault data of the previous pass through the LLP's main program loop. Once all relevant LLPs have responded with quiescent messages, FRAMES will request switch and sensor status from all LLPs to get a *snapshot* of the system after the fault has run its course. Upon sending up the quiescent data, the LLP has finished the task of fault reporting for the fault in question.

2.5 Fault Isolation

Fault Isolation is used by FRAMES as part of diagnosing a probable cause for a given reported fault. When FRAMES is advised of a fault by an LLP, It may be necessary to reclose some of the switches in an attempt to repeat the fault. With this in mind, the LLP accepts commands to turn on and turn off given switches. The LLP immediately takes a new data set from the switchgear and looks for new trips. FRAMES requests new switch and sensor status lists to see if anything new happened. By using this process, FRAMES can make a probable fault diagnosis.

2.6 Redundant Switching

The LLP software is set up to handle redundantly sourced critical loads. If a load is redundantly sourced and the LLP has permission to switch to the redundant switch, the LLP will attempt to repower the load on the redundant bus in the event of a fault. It should be noted that the LLP will not be able to power the load off the redundant source if the LLP cannot find enough power on the redundant bus. The LLP will load shed any load of lower priority than the redundant load necessary to acquire enough power for the redundant load. The software will not shed any loads if it cannot free up enough power for the redundant load.

2.7 Performance Monitoring

The LLP Software monitors the performance of the switches and sensors it controls. For the switches it computes a time based average of current passing through each switch. Furthermore, it keeps track of the maximum and minimum current readings for each switch as well as when they occurred. This computation is run over five minute intervals or until a switching operation occurs. Either of these events will start a new performance interval for a given switch. Before a new performance interval begins, the accumulated data is sent to the SSM/PMAD interface. In the case of the sensors, the data for all current, voltage, and temperature sensors is accumulated and used to compute time based averages for each sensor. This data is sent to the SSM/PMAD interface on a five minute interval.

2.8 Limit Checking

The LLP software performs limit checks on powered loads. The schedule passed to the LLP contains maximum and minimum current limits for a given load. If the load is pulling more current than is allowed, for more than one data accumulation cycle in a row, the LLP will shed the load and inform the SSM/PMAD interface. If the LLP is a PDCU and a switch is reading out of current limits, the LLP software will not shed the switch, but it will inform the SSM/PMAD interface. Shedding a PDCU switch would remove power from the load center bus below it, and this is obviously undesirable.

2.9 Configuration Determination

Upon power up of the LLP and its software, the LLP software determines the configuration of the switches it controls. The LLP queries each SIC card for switch status. The switch status response tells the LLP software how many and what type of switches it controls. With this information, the LLP sets up the appropriate conversion constants for the switch sensors. The LLP software then takes the configuration information and sends it to the SSM/PMAD interface in the form of a configuration list (Appendix V). The SSM/PMAD interface uses this information to represent the LLPs on the user interface.

2.10 Multiline Interrupt Driver

The Multiline Interrupt Driver (MID) by Parasoft was purchased to handle RS-422 serial communications between the LLP and the SIC. The LLP software needed a good serial port driver to handle serial input. The operating system provides a serial port driver, but the driver supplied is inadequate for serial input, because it does not buffer input and data can be lost. The MID software is interrupt driven and buffers the communications at 9600 BAUD. MID is installed on the LLP at bootup, and it is configured as follows:

- A) Number of Ports = 2.
- B) IRQ = 2, on both ports.
- C) User service interrupt = 14H, on both ports.
- D) UART base port addresses = 3E8 , 2E8.
- E) Transmit = 15000 bytes / Receive = 15000 bytes , on both ports.
- F) 9600 BAUD, Even Parity, 8 Bits, 1 Stop Bit , on both ports.
- G) XON/XOFF for both transmit and receive, No hardware handshake,
Short Timeout, on both ports.

2.11 CMC Ethernet Driver

The other driver used by the LLP Software is an ethernet device driver which came with the Communications Machinery Corporation (CMC) ethernet card. The ethernet driver software and ethernet board were selected for this project because they have an assembly level interface capability to their device driver software. Access to this assembly level interface is driven by common software interrupts. This made interfacing with the

LLP software, written in pascal, relatively easy. The CMC software is configured for 2 ethernet channels, 1 Internet Protocol channel, 2 User Datagram Protocol channels, and 4 Transmission Control Protocol Channels. The driver software is also configured for Netbios emulation, and beginning memory address location A0000h.

3.0 LLP Hardware

LLP hardware consists of two distinct types. The first type is the computing hardware platform upon which the LLP software runs. The second type is the power hardware referred to as switchgear. The LLP software commands and controls the switchgear from the computing platform. Both types of hardware have multiple interacting components and a specific configuration which must be maintained in order for the LLP software to function properly.

3.1 LLP Computing Hardware Overview

The LLP computing hardware consists of a Quimax 20 MHz motherboard with 1Mbyte of RAM rehoused into an Integrand rack mount chassis with a 135 watt power supply. This base computer contains five peripheral devices, a 1.2 Mbyte 5 1/4 inch floppy disk drive, a monochrome graphics card (Not Used), a keyboard (Not Used), a Sealevel Systems Dual-SIO dual port RS-422 serial communications card, and a Communications Machinery Corporation ethernet adaptor. The keyboard and monitor are not used in running the full SSM/PMAD system as the LLP software operates autonomously.

3.1.1 LLP Motherboard Configuration

The Quimax 20 MHz motherboard is configured to run only in 20 MHz mode. The motherboard and computer may be reset by pressing the button on the front panel of the rack mount chassis. The motherboard is presently configured to operate without a keyboard, this may be changed by running the setup program upon boot up of the operating system. Only one dip switch and no jumpers have been changed from the default configuration on the motherboard. The motherboard has two serial ports and one parallel port built in. The serial port operating on COM2: has been disabled by setting switch 2 of SW1 to the on position. The serial port needed to be disabled because COM2: uses IRQ 3 which is the same interrupt being used by the CMC ethernet adapter. COM1: which uses IRQ 4 is still available for use. The only other change required for the motherboard is in the Integrand rack mount chassis power supply. The two wires of connector P8 closest to the backplane must be tied together for the computer to boot without a BIOS error. This requires clipping the backmost lead going into the connector and electrically tying the lead

going into the connector to its nearest and only neighbor. This leaves the original wire from the power supply to where it was cut which must be electrically sealed. This change ties power supply ready directly to 5 volts (asserted) as soon as the power supply comes up.

3.1.2 RS-422 Card Configuration

The Sealevel Systems Dual SIO RS-422 serial communications card allows the LLP to communicate using two separate RS-422 ports. The RS-422 card has been modified in order to optically isolate the communication receive data lines. The SIC card provides optical isolation for the transmit lines. By optically isolating the communications, it is unlikely for a power system surge to pass through and damage the computing hardware. Installation of the optical isolation modification requires the removal of the SN75173 IC from the socket at U4 on the RS-422 card. The modification also requires the removal of R11, R15, R16, and R20 from the RS-422 card. The SN75173 IC must be plugged into the optical isolation modification and the optical isolation modification plugged into U4 of RS-422 card. The optical isolation modification schematic is shown in figure 1. The DIP switch and jumper settings are configured as shown in figure 2. The DIP Switches S1 and S2 are used to set UART base port address locations. The jumpers E1 and E2 set the RS-422 card IRQ number for their respective ports; presently they are both accessed on IRQ2.

3.1.3 Ethernet Adapter Configuration

The CMC ethernet adapter is used by the LLP software to communicate with the SSM/PMAD interface. The ethernet card has been configured for IRQ3 by moving the jumper on JP7 from 4 to 3. The ethernet adapter has been configured for IEEE 802.3 thick wire communications. This was done by moving the IC in U69 to U68. All other board jumpers and switches were left in the default factory configuration.

The LLP uses different IRQ numbers for different devices. IRQ2 is used for RS-422

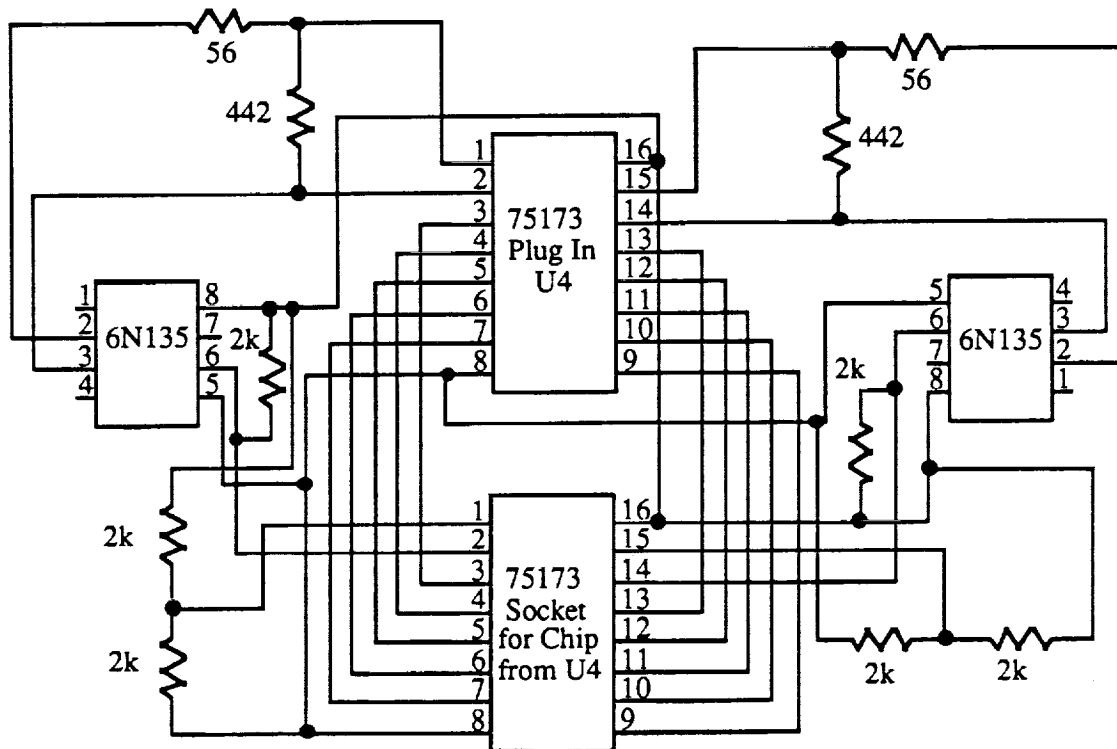


Figure 1 - Optical Isolation Modification

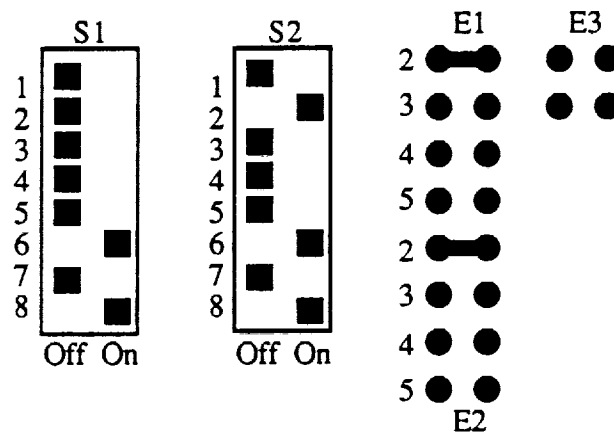


Figure 2 - RS-422 DIP Switch and Jumper Settings

communication and IRQ3 is used by the ethernet adapter. This means the RS-422 card has higher interrupt priority than the ethernet adapter. This was done because the ethernet adapter has its own on board processor and memory and it can therefore buffer communications to a certain extent.

3.2 Switchgear Overview

The LLP switchgear hardware consists of Switchgear Interface Controller (SIC) cards, Generic Controller (GC) cards, Analog to Digital (A/D) card, sensors, and switches. The LLP software communicates only with the SIC card. The SIC card in turn may talk to up to fourteen GCs, but is limited to nine by a card cage constraint. Each GC controls a switch. The SIC may also talk to the A/D card which may read sixteen current, sixteen voltage, and sixteen temperature sensors. In this manner, the LLP software has its data collected by the SIC card on command and returned for processing.

3.2.1 Switchgear Interface Controller

The SIC card responds to a four byte command it receives from its RS-422 receive data lines. To acquire data from the GC cards and their attached switches, the SIC card will assert the GC card in question's data enable line and wait for a two byte response. To accumulate data from the A/D card, the SIC card asserts the start conversion line to the A/D and collects the response in a buffer. The SIC card can be issued nineteen different commands from the LLP and these are described in the LLP/SIC Interface control document (Appendix VI).

3.2.2 Generic Controller

The GC card actually contains two generic controllers on each card. The top generic controller circuit is controlled by the SIC on the right side of the card cage (as seen from the front of the card cage). The bottom generic controller circuit is controlled by the SIC on the left side of the card cage. The GC actually issues an on or off command to the switch. The GC is also responsible for turning the switch off in the event of a trip situation. The GC card handles all trip situations except fast trip. These situations include ground fault, overcurrent, surge current, over temperature, and under voltage. It should also be noted that there are two types of GCs, the older ones which could handle the old AC switchgear,

and those GCs designed specifically for DC.

3.2.3 Switches

There are three varieties of switches: Remote Bus Isolators (RBIs), one kilowatt and three kilowatt Remote Power Controllers (RPCs). The RPCs were designed to break current when commanded off and also contain a self protection fast trip circuit in the event of a dead short across the output terminals of the switch. The RBI was not designed to break current and should always be switched with no current flowing. The RPC/GC combination provides a very good current limiting switch that will trip on a variety of faults. Moreover, the LLP software acquires any trip information when it requests data on the switch.

3.2.4 Sensors

There are three types of sensors; current, voltage, and temperature. The current sensors have 2 ratings, either 50 or 100 amps, whatever is necessary for where they are placed in the topology. The voltage sensors have only one rating; 120 volts. The temperature sensors are rated based upon their thermocouple. All of the sensors return 5 volts at twice their rated value. The A/D card will register a fully set 8 bit word for a 5 volt reading.

4.0 Future LLP Concerns

In the future, it may be necessary to put a flight system together similar to the SSM/PMAD breadboard. Such a system would probably have algorithmic software and smart power hardware on board the spacecraft. The expert systems and artificial intelligence systems would probably be ground based. Communication between the expert systems and the lowest level processors would occur via a telemetry link. This raises several important points: First, by minimizing communication across the telemetry link, expenses of communication will be likewise minimized. Second, the LLPs will need to be able to operate autonomously for large periods of time in the event the telemetry goes down. Third, the reliability of the switchgear hardware must be improved. Lastly, the LLPs or the switchgear they control should be controllable locally on board the spacecraft should it become necessary to take control manually. How to accomplish this capability will now be discussed.

4.1 Feasible LLP Solutions

Communications may be minimized by migrating some of the fault diagnosis capability to the LLPs. This might be done by changing the communications architecture such that a load center reporting a fault would report it first to its controlling PDCU and all appropriate testing be done before shipping the data to the SSM/PMAD interface for diagnosis and rescheduling by MAESTRO. Another issue is how best to improve long term autonomy for the LLPS. A good way to establish long term autonomy at the LLPs is to provide them with longer schedules as well as some amount of schedule recover mechanisms. A longer schedule requires more memory on the LLP. The next issue is improved reliability. The good way to improve reliability in the switchgear is to decrease the number of components. Presently there are around 100 elements of switchgear. Suppose that each of these components has a reliability of 99.5%. When all of these components are integrated together, the reliability of the system would potentially drop to about 60%. Clearly, with fewer components, there are fewer places for failure. One place for switchgear consolidation is the GC/RPC interface, this may be accomplished by building an intelligent RPC with all the relevant GC functionality. Another possible place for consolidation would be the sensors; make the sensors directly readable from the LLP controller. If the LLP is already reading all of the sensor data, make the new intelligent

switch described above directly interfaceable to the LLP controller. If the switches were directly interfaced to the LLP controller, the LLP would be able to control as many switches as could be interfaced. It should be noted that as additional switches are interfaced, the performance of the LLP would degrade to some degree. Making the switches directly interfaced to the LLP controller would eliminate the need for the SIC card. Implementing all of the above suggestions would eliminate the need for A/D, SIC, and GC cards. This would drastically cut the number of components within the switchgear. Moreover, the LLP functionality would not be affected and the reliability would increase. An accessible link to control of the power system should be a requirement so that in the event of a catastrophe, those aboard the spacecraft will have control over the power system. This could be accomplished with a manual control interface directly tied into the LLP functions.

APPENDIX III SSM/PMAD TECHNICAL REFERENCE



SSM/PMAD Technical Reference

Version 1.0

Joel D. Riedesel
Martin Marietta Space Systems
P.O. Box 179, MS: S-0550
Denver, Co. 80201
jriedesel@den.mmc.com



Contents

1	Purpose of this Manual	1
2	SSM/PMAD Breadboard Design	2
2.1	System Design	6
2.2	Detailed Design	9
3	SSM/PMAD Interface	15
3.1	The View of the Breadboard	15
3.2	Overall Control	15
3.3	User Interface Functions	17
3.4	Communications Functions	22
3.4.1	Managing Communication Connections	23
3.4.2	Transactions	26
3.4.3	Utility Communication Functions	49
3.5	Utility Functions	51
4	FRAMES Technical Reference	54
4.1	The FRAMES Architecture	54
4.2	Multiple Faults in SSM/PMAD	55
4.3	The FRAMES Knowledge Base	56
4.3.1	The FRAMES Domain	59
4.3.2	The Hard Fault Expert System	103
4.3.3	The Soft Fault Expert System	138
4.4	Function Reference	141
5	KNOMAD-SSM/PMAD Technical Reference	143
5.1	The Database	144
5.1.1	Tuples and Views	144
5.1.2	Database Constraints	145
5.1.3	Facts and Frames	146
5.1.4	Assertions and Retrievals	148
5.1.5	Locks	149
5.1.6	Initialization	149
5.2	The Rule Management System	150
5.2.1	The Knowledge Base	150
5.2.2	The Rule Group	152
5.2.3	Rule Group Methods	154
5.2.4	Rule Semantics	156



5.3	Adding Tools to KNOMAD-SSM/PMAD	156
A	Suggested Readings	157
B	KNOMAD-SSM/PMAD BNF Syntax	157
B.1	Definitions	157
B.2	Rule Management System	158
B.3	Frames	159
B.4	Database Assertions	160
B.5	Integrity Constraints	160



List of Figures

1	Power System Network Topology	6
2	SSM/PMAD Architecture Hierarchy	7
3	SSM/PMAD Breadboard Control Loops	8
4	SSM/PMAD Breadboard Configuration	10
5	Hardware Communications	11
6	KNOMAD Layered Architecture	143



List of Tables

1	Processes Ports and Hostnames	24
2	*transaction-table*	30
3	*transaction-address*	31



1 Purpose of this Manual

This manual is intended for the maintenance and development of the SSM/PMAD breadboard. It describes in detail the SSM/PMAD breadboard.¹ This manual assumes the reader is familiar with the SSM/PMAD breadboard and has at least read appendix I of this report, the SSM/PMAD Interface User Manual. Section 2 describes the overall design of the breadboard, the various functions of it, and how they interact with each other. The partitioning of algorithmic (and AI) functions to hardware components, whether circuit boards or computers, is also discussed. In some cases, the reason for the placement of a particular function on a piece of hardware is not absolute but rather, is in terms of convenience.

The design of the SSM/PMAD breadboard includes many functional components including the LLPs, FELES, LPLMS, MAESTRO, FRAMES, and KNOMAD-SSM/PMAD as well as hardware functions. Each of these will be discussed in section 2 as part of the overall design. The manner in which these components interact with each other is also of primary importance and has been described as three nested control loops in one paper [RMA]. These interactions will also be described.

Section 3 will provide a detailed description of the design and implementation of the SSM/PMAD interface. This includes descriptions of the user interface, communications, control operations, and utility functions. The SSM/PMAD interface provides a model of the SSM/PMAD breadboard. By understanding the SSM/PMAD interface, the system designer will understand how the various components are integrated together in a conceptual perspective as opposed to a functional level as described in section 2.

Section 4 provides a detailed description of the FRAMES knowledge base as well as associated algorithmic functions to support it. This section will describe the expert systems and the domain and possible modifications.

Section 5 describes the design and implementation of KNOMAD-SSM/PMAD. It includes a description of the database, the database interface, the rule management system and the usage of these elements.

Appendix A of this appendix provides a list of suggested readings for information about both the SSM/PMAD breadboard and KNOMAD-SSM/PMAD.

Appendix B provides the designer with the current syntax of KNOMAD-SSM/PMAD.

¹This work was performed under contract NAS8-36433 to NASA, Marshall Space Flight Center.

2 SSM/PMAD Breadboard Design

The SSM/PMAD testbed, an abbreviation for Space Station Module Power Management and Distribution, is a project of the Electrical Power Systems group at NASA, Marshall Space Flight Center. The goal of the testbed is to build a breadboard containing both hardware and software components for researching techniques to automate a power management and distribution system that could be applicable to Space Station Freedom. This document will describe Version 1.0 of the SSM/PMAD Breadboard. This version could be considered as the second implementation. The first implementation both achieved the goal and proved the concept but was not very robust.² Version 1.0 tremendously improves the robustness of the breadboard, while also adding functionality and ease of use.

The autonomous power management and distribution problem consists of the following requirements:

- Switches that control power to loads.
- The ability to schedule activities using power as a resource and have the schedule be executed by the power system to enable the loads.
- The ability to detect, diagnose, and recover from power system faults.
- The ability to operate autonomously.

These very high level requirements of the SSM/PMAD testbed make the problem sound fairly simple, but the design is actually quite complex, involving a large amount of both hardware and software.

Given the stated goal and requirements, the SSM/PMAD breadboard has gradually evolved to its current architecture. The architecture of the breadboard is the topic of this section and will be described in detail in the following two subsections. A system level design description will be given followed by a subsection which goes into detail explaining how the components are integrated into the completed breadboard.

As an overview of the functionality of the breadboard, the rest of this introduction to the breadboard design will describe the functionality of the hardware and software components that were necessary for the SSM/PMAD testbed to meet its goal. These components range from software utilizing artificial intelligence techniques to advanced development of hardware components for remotely operating switches controlling power.

The functional components of the breadboard can be described in three layers³. The first layer, where power is shunted through switches, is the switchgear layer—composed entirely

²See [MJA*] for a description of the first delivered implementation.

³These layers are referred to in the following sections as the first, second, and third layers, or as the bottom, middle, and top layers, respectively.

of hardware. The functional components (from a system operations perspective) of this layer are:

Switches Various types of switches detect and control against a low-impedance shorts in the power system. These switches are remotely operatable, receive commands to open or close, and transmit trip data. Since the design of the Space Station Freedom power system was not finalized at the beginning of the contract, there was a need for both DC and 20KHz AC switches.

Switch Controllers In order to operate different types of switches, a generic controller was developed. This controller identifies different types and characteristics of switches and determines over-current, ground fault, under voltage, over temperature, and surge current trips. In the case of these trip conditions, for a particular switch in question, the controller collects data from the switch, identifies the trip condition, trips the switch off, and transmits the trip data. The controller receives commands and transmits switch data and trip information.

Sensors The ability to make intelligent decisions about an artifact is directly related to the amount of information that is available from it. The switches, through the switch controllers, supply some information about the power system. Sensors are also required to provide further information. This is comprised of additional information not available from the switches. Sensors also provide the ability to determine soft faults (an illegal use of current, probably the result of a high-impedance short to ground somewhere in the power network). Sensors transmit their sensed data to an A/D card for conversion and transmission.

Analog to Digital Conversion Analog to digital converters provide the means to interpret analog sensor data and provide it to other components for analysis.

Interface Controller The final hardware component necessary to operate a complex set of switchgear is the interface controller. This component interfaces between the lowest layer of software and the switchgear. It provides the ability for the software functions above it to receive data from switches and sensors as well as command switches on and off. The interface controller may interface to many individual switch controllers, providing a level of multiplexing.

The second layer of functional components is the set of software components that provide the deterministic algorithms for the SSM/PMAD breadboard. These are:

Schedule Execution A schedule of switch commands for enabling the loads of a scheduled set of activities is used to command the switches open and closed. This simple schedule execution function must maintain an accurate representation of clock time. A schedule

consists of a list of events, while each event specifies what switch to command, the particular command, and maximum and minimum amperage and power limits for the switch.

Performance Monitoring In addition to commanding switches open and closed in response to a schedule, performance monitoring is used to track the power, voltage, and amperage of switch and sensor data. This data is used by other functions as well as being averaged for a general utilization function by which the operator may observe power utilization of the system.

Fault Detection A function for detecting that a switch has tripped must be available to inform the fault diagnosis function of the fault. This function can detect any of the types of trip a switch can trip or be tripped on.

Limit Checking A schedule provides limits on amperage and power utilized through a switch. Therefore, a limit checking function must exist to make sure that the power and amperage being utilized through a switch is within the set bounds.

Load Shedding When a switch is consuming more than allowed, it may be commanded off and the load below it shed. This function enforces the rule that user's loads follow their specification, and as soon as they operate out of bounds, the load may be shed.

Redundant Switching If there is contingency (e.g., when a switch trips), an important load may need to switch to the redundant power supply. This information is present in the schedule. A redundant switching function must exist to perform this operation when necessary.

Fault Isolation To support the fault diagnosis function, fault isolation must be performed. This is a function that conceptually resides at both the second and third layers of functionality. Fault isolation is performed by both the fault diagnosis function and supported at the second layer by the low level fault isolation function which commands switches on and off. This is entirely analogous to the schedule execution command but used for fault isolation.

The third and final layer of components consists of the set of functional software components that provide the high-level intelligent control to the breadboard. These are functions such as scheduling and fault diagnosis. They are:

Schedule Interface For the operator to specify a set of activities to be scheduled, an interface must be provided. This function is used to specify activities and define equipment and their connections to the power system resources. The interface also provides the ability for the operator to schedule the selected set of activities.

Schedule The schedule function is a high level tool that takes a set of activities, their equipment and a set of available resources as input and produces a schedule for the subset of activities that makes efficient use of the resources. This function enables the operator to efficiently produce schedules for a Space Station Freedom environment without being burdened by too many options.

Operating Interface The operator needs to be able to interface with the breadboard. This is accomplished via the main interface to the breadboard. This interface provides the operator actual control of the breadboard, whether the breadboard is being operated manually or autonomously. The operator is able to manually command switches and observe data from the switchgear using this interface.

Schedule Conversion A schedule as produced by the schedule function must be converted to a schedule of events for commanding switches open and closed. This process includes collecting information about operating limits for the activities and including that in the schedule of events as well (for example, maximum and minimum amperage limits, redundant power sourcing, etc.).

Priority Generation The functions of load shedding and redundancy switching involve possible shedding of loads of lower priority. The priority generating function is responsible for determining priorities on loads (and thus, switches) that are in synchronization with the schedule. The load shedding and redundant switching functions use the generated priority list to help determine how to perform their respective function.

Fault Diagnosis The fault diagnosis function is responsible for collecting data from the fault detection function and perform isolation to determine a diagnosis. A diagnosis is one part of the fault detection, diagnosis and recovery process of a robust autonomous system.

Fault Isolation The fault isolation function at this level is responsible for generating a list of switch commands for the fault isolation function at the second layer. Fault isolation uses the results from commanding the switches to help with the fault diagnosis.

Fault Recovery Fault recovery is used after fault diagnosis to inform the scheduler of switches that are no longer available in the power system. The scheduler function will then reschedule activities based on this new information to develop a new schedule that making efficient use of the remaining resources.

The above functional requirements of the SSM/PMAD breadboard provide a very high level description of the entire breadboard and what its necessary components are. The next two subsections will describe the design of the breadboard in much greater detail including the actual components designed under the contract.

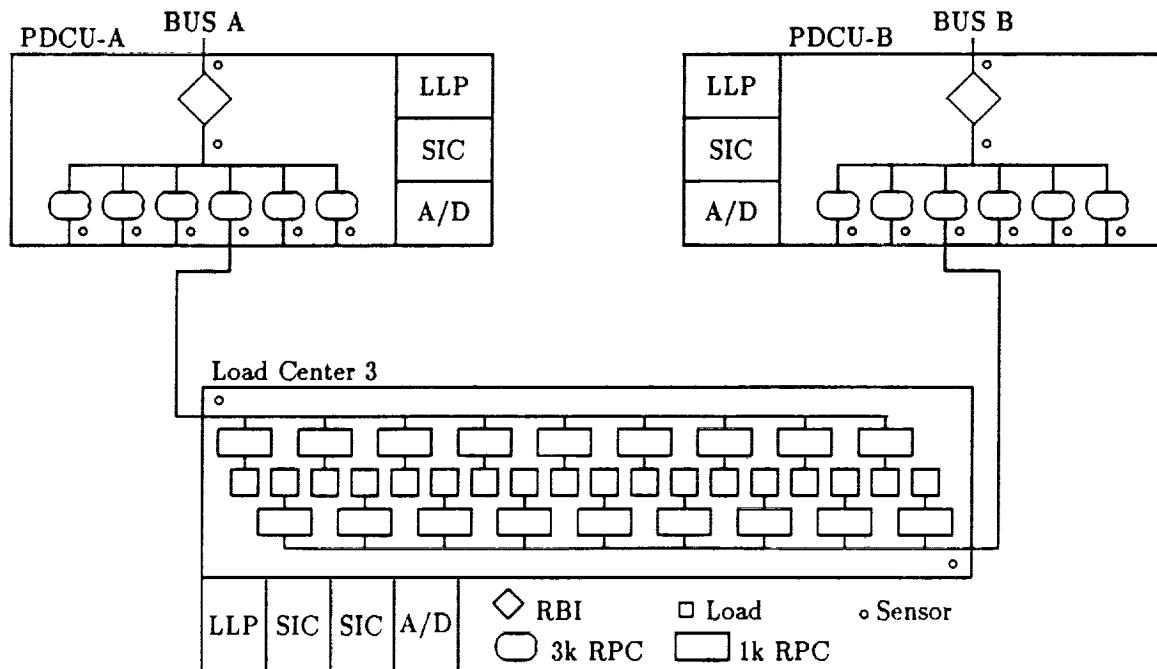


Figure 1: Power System Network Topology

2.1 System Design

The SSM/PMAD breadboard is a testbed to model the Space Station Freedom power management and distribution system. The most distinguishing feature of the breadboard is the power system. The power system topology is shown in figure 1⁴. This topology shows a star bus configuration where each power bus is distributed through a power distribution control unit and to a number of load centers. The figure shows the first layer of functional components of the breadboard and how they are interfaced to the power system.

The power system topology implements the first layer of components by designing and building the various hardware pieces to meet the first layer of functional requirements. These elements are shown in the figure and described as follows. The switches are one kilowatt, three kilowatt, and 15 kilowatt remotely operable devices. The RBI is a remote bus isolator and is a relatively simple switch that can be remotely commanded on and off but will not trip. The 1K and 3K switches are remote power controllers (RPCs) that are *smart* switches. They will trip on a low-impedance short. The switch controller is a generic controller (GC) (not depicted in the figure) that adds further logic to the switches. In the case of an RPC, it will detect high-impedance shorts that will result in an i^2t trip as well as low voltage situations

⁴The figure only shows one load center; as can be seen from the PDCUs, however, there may be up to six load centers.

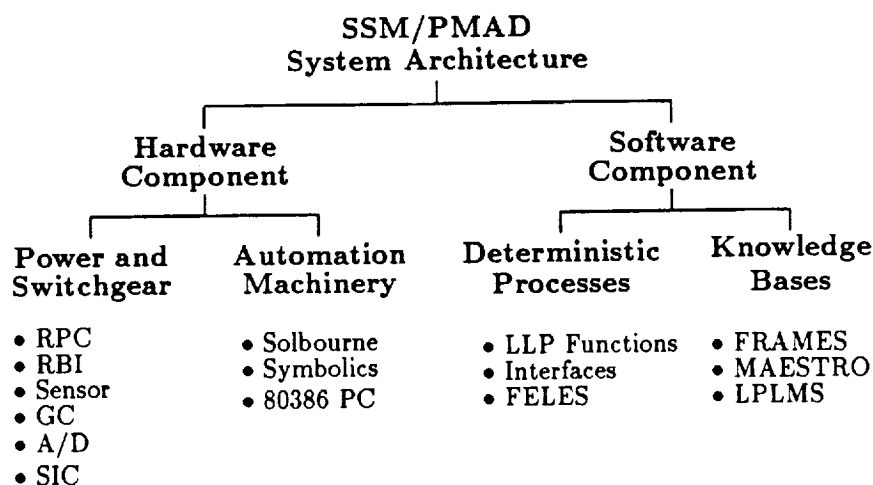


Figure 2: SSM/PMAD Architecture Hierarchy

that will result in an under voltage trip. The GC allows both AC and DC switches to be controlled. Version 1.0 of the breadboard is implemented with only DC switches.

Sensors are depicted in the figure and are implemented in different sizes depending upon the rating of the sensed current. Current sensors are either 50 or 100 amps. All voltage sensors are 120 volt DC sensors. The analog to digital converter is shown in the figure and is used to collect analog data from the sensors and digitize it.

The switchgear interface controller (SIC) allows communications between the hardware and upper layer software components (shown in the figure). There is one SIC for each bus of a load center and one SIC for each PDCU.

The LLP shown in the figure is a lowest level processor. The LLP implements the functions of the second layer and is currently based upon an 80386 PC clone.

In summary, the figure shows the power system network topology as well as the LLP which implements the second layer functions. The power system topology is interesting because it is distributed. Each load center or power distribution control unit is operated by a different LLP. This gives the power system some robustness such that if part of the power system fails, the rest may keep operating. The power system and switchgear hardware components are described in detail in [And].

To continue illustrating the design of the breadboard, figure 2 shows the partitioning of the functional components to hardware and software. The power and switchgear hardware was discussed above in relation to figure 1. The other hardware consists of the computer systems used to execute the software functions of the breadboard. These are the 80386 PCs, the Symbolics, and the Solbourne. The LLPs (implemented on the 80386 PC) are used to implement the second layer functions described earlier which include, for example, schedule execution, performance monitoring, fault detection, and limit checking. The Solbourne and

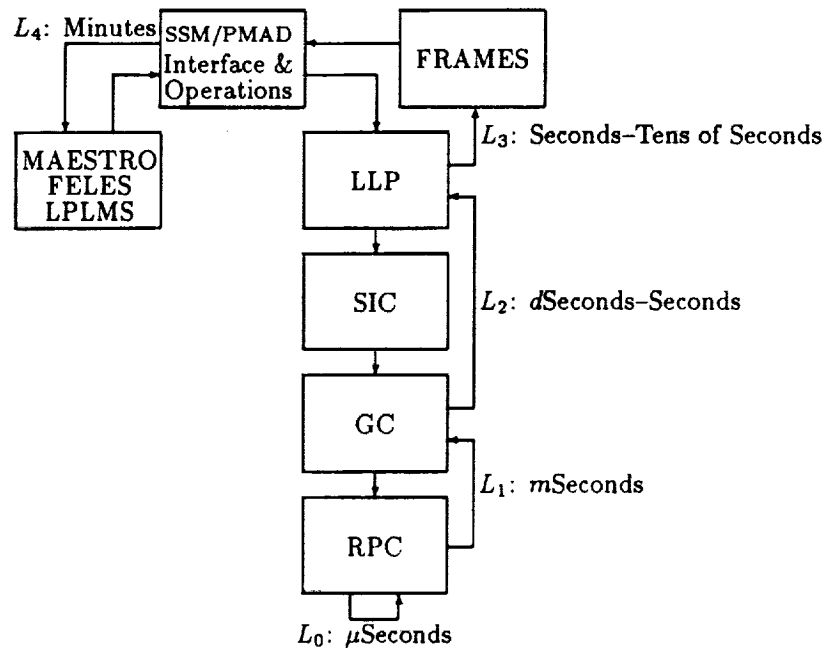


Figure 3: SSM/PMAD Breadboard Control Loops

the Symbolics are used to implement the third layer functions including the interfaces, the scheduler, and the fault diagnosis system.

The other objective of figure 2 shows how the second and third layer functions are partitioned into deterministic and knowledge-based implementations. The LLP functions are all those second layer functions described earlier. The interfaces, both to the scheduler and the SSM/PMAD breadboard, are considered deterministic algorithms. FELES is the Front End Load Enable Scheduler which performs the necessary schedule conversion function of the third layer.

The knowledge-based functions are all functions of the top layer and currently consist of FRAMES—the Fault Recovery and Management Expert System, MAESTRO—the scheduler, and LPLMS—the Load Priority List Management System.

The general operation of the SSM/PMAD breadboard, in terms of general data flow and control loops, is shown in figure 3. The figure is intended to show that operation of the breadboard is controlled by the SSM/PMAD interface. A schedule is created by MAESTRO, converted by FELES to a list of switch commands, and is made available to the SSM/PMAD interface. The schedule of switch events, along with a priority list from LPLMS, is given to the LLPs. The LLPs then command switches on and off via the SIC.

There are five control loops shown in the figure. Control loop L_0 is contained entirely in hardware on the RPC. This control loop monitors current through the switch and will

trip the switch on a fast trip if a low-impedance short is detected. This is the only real-time control loop and occurs in the region of microseconds.

Control loop L_1 occurs between the GC and the RPC. In this loop, the GC monitors both current and under voltage. The GC will trip the switch on ground fault, over current, or voltage depending upon the data from the switch. This loop occurs in the region of milliseconds.

Control loop L_2 occurs between the LLPs and the RPCs. This loop is responsible for limit checking, load shedding, redundancy switching, and the functions of the bottom layer. This control loop can operate from tenths of seconds to seconds. The main performance problem is that the A/D boards of the switchgear require a 1.5 second delay every time the LLP needs to get data from them through the SIC.

Control loop L_3 occurs between FRAMES and the switchgear. This is the loop for performing fault isolation and diagnosis. As faults occur in the breadboard, FRAMES takes over and performs fault isolation by commanding switches open and closed in order to isolate the location of the fault. Once the fault has been determined a diagnosis is made. This control loop operates in the seconds to tens of seconds range. Commands as a part of fault isolation can be executed in seconds, but a fault diagnosis may involve a sequence of fault isolation steps resulting in performance on the order of tens of seconds.

Control loop L_4 is the loop between the scheduler and the rest of the breadboard. This loop is executed whenever a contingency occurs in the breadboard and rescheduling needs to take place. At that time, FRAMES will inform the scheduler of the unavailable resources and MAESTRO will perform contingency scheduling to make efficient use of the remaining available resources. Control loop L_4 operates within minutes.

This completes the high level description of the design of the SSM/PMAD breadboard. The next subsection will describe the integrated design of the system that includes a discussion of data communications between the components of the breadboard.

2.2 Detailed Design

Figure 4 shows the integration of the components of the SSM/PMAD breadboard. The user interacts with the breadboard through the primary interface, the SSM/PMAD interface. When scheduling, the user interacts with the scheduler interface which resides on the symbolics. The user manual (appendix I of this report) describes how to use the breadboard.

The hardware is organized such that the SIC communicates to the RPCs and RBIs via the GC cards. The SIC also collects sensor data via the A/D board. The second layer of functions is implemented on the LLP using the 80386 PC environment. The LLP is used to interface to the hardware via the SSM/PMAD interface.

The third layer of functions is partitioned to the Symbolics and the Solbourne. MAESTRO, the scheduler, is implemented on the Symbolics as is FELES and LPLMS. The

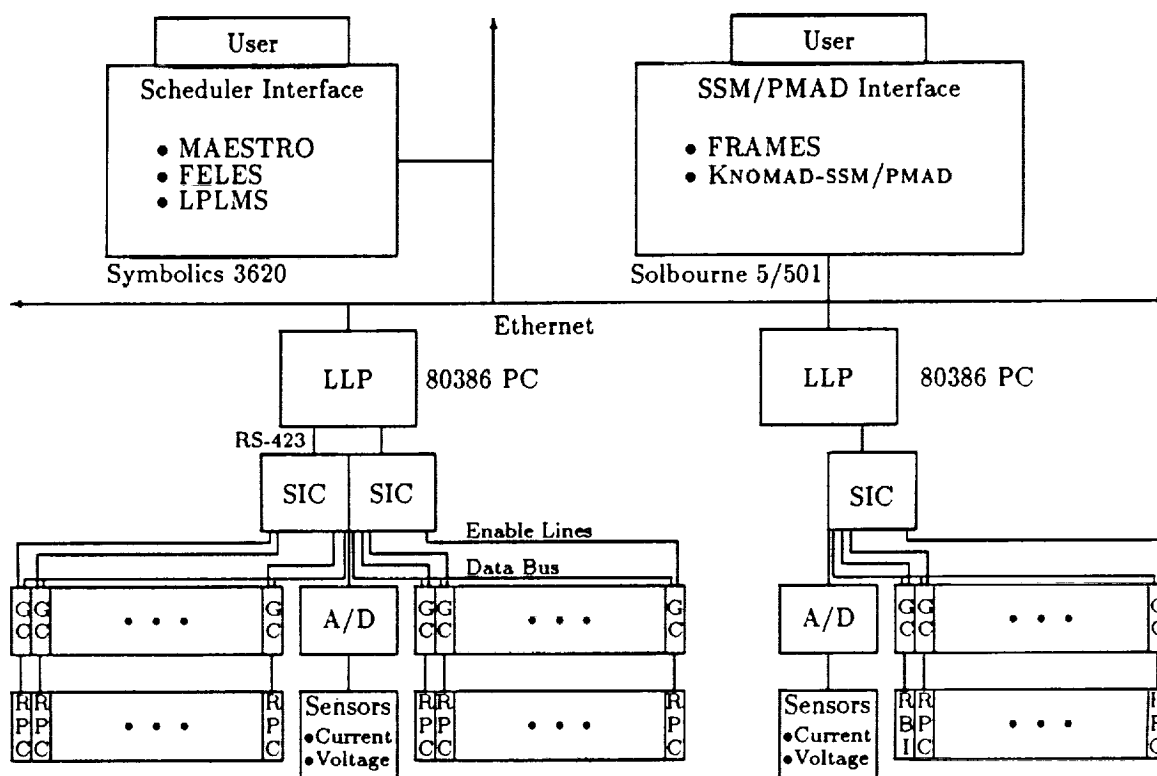


Figure 4: SSM/PMAD Breadboard Configuration

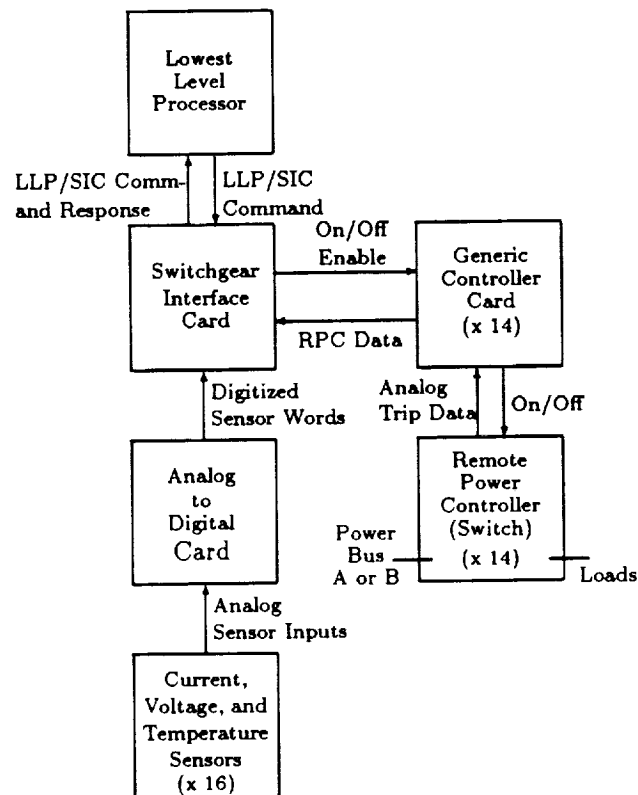


Figure 5: Hardware Communications

SSM/PMAD interface and FRAMES is implemented on the Solbourne computer. KNOMAD-SSM/PMAD is used as the support environment for FRAMES and the SSM/PMAD interface.

The higher level functions, second and third layers, communicate using TCP/IP over ethernet. The LLP communicates to the SIC using RS-423.

Figure 5 provides a representation of the hardware components and their integration with one another. The LLP may send a command to the SIC to retrieve data from the A/D card, or from the GC card. The LLP may also command a switch via the GC. Data is sent back to the LLP from the SIC. The interface control document for the LLP and SIC communication is given in appendix VI of this report.

The LLPs do not perform any functions without direction from the SSM/PMAD interface. When the SSM/PMAD breadboard is being operated, the SSM/PMAD interface sends commands to the LLPs and collects data from them. These transactions include obvious things such as switch data, sensor data, and event lists. When a user requests data on a switch, the SSM/PMAD interface sends a switch data query to the LLP and the LLP responds with a switch data transaction. When a user sends a request to turn on a switch, the SSM/PMAD interface sends a switch control list to the LLP. The LLP then responds

with a switch and sensor configuration transaction. The transactions that are used between the SSM/PMAD interface and the LLPs are given in appendix V of this report. However, appendix V is simply an ICD and doesn't describe the semantics of the transactions being passed back and forth.

To start the breadboard, the SSM/PMAD interface sends an initialization transaction to the LLPs. When the LLP initializes, it resets its internal states about the configuration of the switches. The LLP will respond with switch and A/D conversion values lists and the switch sensor configuration. The second step of starting the breadboard is to send a time list to the LLPs. This sets the clock on the LLPs and finishes their initialization.

The LLP may be queried for many types of information to which it will respond. If a switch is commanded on or off, the LLP will respond with a switch configuration list to inform the interface of the new state of the switches. Furthermore, the LLP monitors performance data of the switches and sensors and periodically sends up switch and sensor performance lists.

To summarize, the following provides a brief overview of the semantics of the various transactions between the LLPs and the SSM/PMAD interface:

Initialization List Sent by the SSM/PMAD interface to initialize the LLP. This must be followed by a time list to finish the LLP initialization. The LLP will respond with three lists when it receives an initialization: A switch conversion value list, an A/D conversion value list, and a switch/sensor configuration list.

Time List Sent by the SSM/PMAD interface to the LLP to finish the initialization of the LLP. The time list consists of the current time and the start of mission time. In the case of manual mode, mission time is the same as the current time.

Event List The event list is a list of switch control events for commanding switches on and off. This is sent from the SSM/PMAD interface to the LLP when the system is in autonomous mode. In Version 1.0 of the breadboard, the event list provides a thirty minute block of switch events for the LLP to use.

Priority List The priority list is sent from the SSM/PMAD interface to the LLP. This list is used by the LLP when it needs to shed a load. It provides the information on which loads should be shed first.

Contingency Event List This is sent to the LLP by the SSM/PMAD interface. It is used when a contingency has occurred in the breadboard to primarily make sure the states of all the switches are in the correct positions, and secondarily, to provide the next event list used by the LLP.

Switch Control List Sent to the LLP by the SSM/PMAD interface. This is used in manual mode to turn switches on and off. The LLP will respond with a switch configuration list to let the interface know of the updated state of the switch.

Switch Conversion Constant List Sent to the LLP by the SSM/PMAD interface to change the LLPs switch conversion constants.

A/D Conversion Constant List Sent to the LLP by the SSM/PMAD interface to change the LLPs A/D conversion constants.

Query List Sent to the LLP by the SSM/PMAD interface to request information. Various types of information may be requested including switch and sensor status, conversion value lists, and the configuration list.

Switch Performance List Sent from the LLP to the SSM/PMAD interface. This list is used to provide switch performance data to the interface. This is a time averaged amperage usage for the switch.

Sensor Performance List Sent from the LLP to the SSM/PMAD interface. This list is used to provide sensor performance data to the interface. This is a time averaged power, voltage, and amperage usage for the sensors.

Switch Status List Sent by the LLP to the SSM/PMAD interface. This transaction provides detailed data about the switches to the interface.

Sensor Status List Sent by the LLP to the SSM/PMAD interface. This transaction provides detailed data about the sensors to the interface.

In addition to these ordinary transactions, there are some transactions used primarily for the purpose of collecting data for fault diagnosis. This is termed as collecting a *snapshot* of the power system (this snapshot collection procedure is fairly complex due to the distributed nature of the LLPs). The current procedure tries to synchronize the data from LLPs as much as possible based upon the speed of the Solbourne and the speed of the ethernet. The procedure is as follows:

1. When an LLP first detects a fault in the switchgear, it sends a fault transaction to the SSM/PMAD interface.
2. The SSM/PMAD interface then sends a quiescent transaction to each LLP.
3. Each LLP responds with a *quiescent is true* transaction when their data is in a stable state.
4. The SSM/PMAD interface then sends a query for switch status information to each LLP.
5. Each LLP then sends the switch status transaction as requested.

6. The SSM/PMAD interface then checks to see if the snapshot of the power system it has just collected is stable. If an LLP has had new fault data during the data collection stage, it will set a non-quiescent bit in the switch status transaction. If the snapshot is not stable, the SSM/PMAD interface will repeat steps 2-5 until a quiescent snapshot is achieved.

The user manual (in appendix I) provides more information on how transactions are sent back and forth in the system.

The last piece of integration in the SSM/PMAD breadboard is between the Symbolics and the Solbourne. This involves communications between MAESTRO, FELES, LPLMS and the SSM/PMAD interface. However, this is only needed if autonomous mode will be used in the breadboard. If this is the case, event list, priority list, and contingency data must be passed between the Symbolics and the SSM/PMAD interface.

When the breadboard is being initialized and autonomous mode is being started, the SSM/PMAD interface must initialize FELES. Initialization of FELES will direct FELES to send a schedule and priority list back to the SSM/PMAD interface which can then distribute it to the LLPs. When the SSM/PMAD interface gets the start of mission time from the user, it is sent to FELES so that it can synchronize with the rest of the breadboard. Autonomous operation periodically sends a schedule (for a period of time) from MAESTRO through FELES to the SSM/PMAD interface. The priority list generated by LPLMS is also periodically transmitted. When a contingency occurs, the SSM/PMAD interface sends a contingency start transaction to FELES. Any load sheds, switch to redundants, or out of services are sent to FELES followed by a contingency end transaction. FELES then directs MAESTRO to generate a contingency schedule. When MAESTRO finishes that task, FELES sends the new schedule, a contingency event list, and the new priority list to the SSM/PMAD interface and normal operations resume.

In summary of the system design of the SSM/PMAD breadboard, the integration of the various components is very complex and a large number of transactions are implemented to keep the system operating correctly. The system cannot be understood simply in this overview which talks about system design and integration; each component must be examined in detail to understand the necessity of the various transactions and their impacts on the various components.

3 SSM/PMAD Interface

The SSM/PMAD interface to the SSM/PMAD breadboard is documented in appendix I of this report, the SSM/PMAD Interface User Manual. That document describes the interface and the operation of the breadboard; here the SSM/PMAD interface is described from a technical point of view.

3.1 The View of the Breadboard

The SSM/PMAD interface provides the managing environment for the operation of the breadboard. It controls how the user interacts with the breadboard, through the user interface. In so doing, the user's conceptual model of the breadboard is modified by what the interface provides. Therefore a *good* interface will strive to provide both an accurate representation of the artifact being interfaced to as well as provide a certain amount of simplicity and abstraction to the artifact.

For the SSM/PMAD breadboard the user operates a complex environment of hardware and software. The hardware consists of over 100 components that must be correctly operated. The software is likewise complex and must be operated in an integrated fashion to properly control the breadboard. When the user desires to turn a switch on, the interface should provide a conceptual model of turning a switch on that allows the user to do so in a straightforward manner. The SSM/PMAD interface allows this by simply representing the switch on the user interface. The user may then *click* on this switch to perform actions on it, thereby manipulating the hardware and software components of the breadboard. The intricate details of the system have been abstracted and simplified so that the user's conceptual model of the breadboard is both accurate and simple.

This technical document will go a step further and describe the details of the SSM/PMAD interface so that a system developer will have the tools available to make intelligent modifications to the interface.

3.2 Overall Control

The SSM/PMAD interface manages the entire SSM/PMAD breadboard, whether the user operates it in manual or autonomous mode. This primarily involves the operation and control of the software components of the breadboard. On the Solbourne, the SSM/PMAD interface consists of the control functions for starting, operating, and stopping the breadboard. It also uses the user interface module of the breadboard extensively to keep the user informed of the state of the breadboard. To provide the necessary integration between the various software components of the breadboard that reside on different physical computers, an extensive set of communication tools are also used by the SSM/PMAD interface.

The primary mechanism whereby the user is presented with the model of the breadboard is the user interface. The user interface functions are described in the next sub-section. The communication functions are described in the following sub-section. The communication functions are a semi-intelligent set of functions that update various other modules of the SSM/PMAD interface automatically. This includes updating the user interface when a switch state changes, calling a control function when a fault is detected so that fault diagnosis can be performed, and always updating the database that is used by all the modules of the interface. The last sub-section of this section describes a few utility functions used by various parts of the interface.

The rest of this sub-section describes the control of the breadboard and the functions used.

The SSM/PMAD interface relies heavily on the internal representation of the breadboard. This representation is the domain and is defined in the `frames.domain` file. The fault diagnosis expert system, FRAMES, requires the domain file as part of its definition and is the initial reason for having it in the first place. However, `KNOMAD-SSM/PMAD` provides an integrated environment for a large number of modules/agents to work together and share data in a controlled fashion. The domain also provides for the sharing of data between the components of the breadboard in an organized fashion by taking advantage of `KNOMAD-SSM/PMAD`. `KNOMAD-SSM/PMAD` will be discussed in a later section as well FRAMES.

Before the SSM/PMAD interface can be started by the user `KNOMAD-SSM/PMAD` must be loaded. This is done automatically for the user. The interface is then started in earnest by the loading of the domain file. The last statement of the domain file is a call to the `run-frames` function.

`run-frames` [Function]

`run-frames` initializes the internal representation of the state of the breadboard and initializes the user interface. It also starts the internal clock for the interface.

At this point the user will be presented with the user interface for the breadboard. In terms of operating the breadboard, the only thing that has been done by `run-frames` is to initialize the SSM/PMAD interface. The user must now initialize frames.

`initialize-frames how` [Function]

`initialize-frames` takes one argument *how*. This argument may have one of four values: `:both`, `:bus-a`, `:bus-b`, `:none` and is used for initializing the switches of the PDCUs accordingly. If *how* is `:bus-a`, when PDCU-A starts communicating with the interface it will be commanded to turn on all its switches; however, PDCU-B will not be so commanded.

Initializing the breadboard with this function also starts up some important background processes as well as bringing the user into manual mode.

When the user is initialized and operating the breadboard, the user may switch between autonomous and manual modes of operation. Initially the user will be in manual mode and may move to autonomous mode.

autonomous-mode

[Function]

tt autonomous-mode performs two important functions: The first function is to reset the breadboard according to how it was initialized. That is, the PDCUs will be initialized according to the *how* parameter of the initialize frames function. All the other load center switches will be shut off. Autonomous mode of the breadboard is defined to work in a well-defined state. When the user chooses to go into autonomous mode, the user is really saying that the breadboard is going to be operated according to the schedule generated by MAESTRO. The user forfeits any manual operations by making this choice.

The second important function is to start FRAMES. The fault diagnosis expert system is loaded and initialized by going into autonomous mode.

When these two functions are completed, the symbolics is initialized and the user is prompted for a time to start the mission.

manual-mode

[Function]

manual-mode may be selected at any time by the user when the system is in autonomous mode. This function puts the breadboard back into manual mode of operation by clearing the schedules that the LLPs are operating and by stopping FRAMES.

When the user is finished using the breadboard and is ready to bring the SSM/PMAD interface down the stop function is used.

stop-frames

[Function]

stop-frames is responsible for stopping all the background processes of the SSM/PMAD interface as well as writing out log information to the various archive files. These archive files will track every run of the SSM/PMAD interface by archiving all communications to and from the interface and all messages to the status window and diagnosis window.

3.3 User Interface Functions

The SSM/PMAD user interface provides the user with access to the breadboard. It includes various functions accessible from the menus as well as functions accessible from the objects on the user interface themselves. The user interface is initialized using the `ui::run-frames` function.

ui::run-frames [Function]

This function, in the **ui** package, creates the user interface. If for some reason the user interface becomes unusable, it may be possible to call this function to make a fresh one.

ui::make-frames-title-menus [Function]

ui::make-frames-title-window [Function]

ui::make-title-window [Function]

ui::make-schematic-window [Function]

ui::make-status-window [Function]

These functions are all called from the **ui::run-frames** function. As their names imply, they are for making the various windows and menus of the user interface.

ui::run-frames1 [Function]

This function is the last function called by **ui::run-frames**. It may be thought of as a continuation of **ui::run-frames**. **ui::run-frames1** does all the work of defining the components of the schematic window, that is, the LLPs, the RPCs, etc.

ui::llp-window [Function]

ui::define-sensors [Function]

ui::define-rpc [Function]

ui::define-loads [Function]

ui::define-cables [Function]

These functions are used by **ui::run-frames1** to define the components of the user interface.

ui::draw-rpc *rpc how* [Function]

ui::draw-cable *cable how* [Function]

ui::draw-load *load how* [Function]

These are the primitive user interface drawing functions. They are used to keep the interface up to date with the state of the breadboard. The *how* argument is the same in all the functions, its value can be one of: **:hollow**, **:filled**, **:invisible**, **:faulted**, and **:out-of-service**. *rpc* is a symbol for the name of an rpc, **a03**, or **c05**, for example. *load* is a symbol for the name of a load, and *cable* is a symbol for the name of a symbol.

<code>propagate item</code>	[Function]
<code>update-switch rpc command &key :tripped</code>	[Function]
<code>update-cable cable</code>	[Function]
<code>update-lc-cables llp</code>	[Function]
<code>update-load load</code>	[Function]

These functions are used to update the power system domain. When a transaction comes from an LLP to the Solbourne describing the switch configuration or switch status these functions are used to both update the database to keep the domain up to date and to update the user interface to reflect the new switch state and power flow status through the cables.

item is the symbol for a switch or load but not a cable. The `update-switch` function takes two required arguments and one optional keyword argument. The *rpc* argument is a symbol for a switch. *command* can be one of `:open`, `:closed`, and `:faulted`. If `:faulted` is used it is a good idea to use the keyword argument to specify the type of trip, e.g. `:fast-trip`.

<code>make-available-llp llp</code>	[Function]
<code>make-unavailable-llp llp</code>	[Function]
<code>make-available-switch rpc</code>	[Function]
<code>make-unavailable-switch rpc</code>	[Function]
<code>make-available-sensor sensor</code>	[Function]
<code>make-unavailable-sensor sensor</code>	[Function]
<code>make-available-load load</code>	[Function]
<code>make-unavailable-load</code>	[Function]

These functions are used to update the availability of the various components of the bread-board power system. They will both update the database of the domain as well as update the user interface. Unavailable components on the user interface will not be accessible or manipulable by the user.

<code>*status-win*</code>	[Variable]
<code>*diagnosis-win*</code>	[Variable]
<code>*prompt-window*</code>	[Variable]

These variables are bound to windows for displaying data about the breadboard. The `*prompt-window*` is used to display one line messages to the user. The other two windows,

status-win and ***diagnosis-win*** are used to display status messages and diagnoses. These windows are only visible if the user decides to view them.

print-prompt-window *message* [Function]

This function is used to display messages in the ***prompt-window***. *message* should be a string, probably not more than about 40 characters, that will get printed in the window.

make-status-win *&key :lines-high :chars-wide :title* [Function]
:background-color :foreground-color :frame-color :left
:bottom :borders :font :title-font

make-status-win is a function used to make a scrollable text window. This window may be customized quite extensively by using the keyword arguments. Arguments such as *:background-color*, if given, must be a legal common windows color.

:lines-high defaults to 20, *:chars-wide* to 80. The background color will default to white while the foreground color and the frame color will default to black. There is no default title (it defaults to *nil*). The position defaults to a left of 100 and bottom of 100. The borders default to 2 pixels. The *:font* defaults to ***system-font*** (a common windows variable), and the title font defaults to ***title-font***.

The window, once created, can then be manipulated with the following functions to display it, to print strings in it, and to kill it. The window will be a scrollable window and allow scrolling when there are more lines of text in it than fit in the window. **make-status-win** returns a *wid*.

status-win-write-line *wid line* [Function]

exposing-status-win-write-line *wid line* [Function]

status-win-write-lines *wid &rest lines* [Function]

These functions are used to write text strings to a window as created by **make-status-win**. *wid* should be the item returned by **make-status-win**.

status-win-write-line takes a *wid* and a *line*. The *line* should be string of not more than the character width of the window. **exposing-status-win-write-line** functions identically to **status-win-write-line** except that if the window is scrolled to the bottom, the window will be automatically exposed so that the user can see the new message in the window. The **status-win-write-lines** function is also analogous to **status-win-write-line** except that any number of lines instead of just one may be passed to it. The function will then use **status-win-write-line** to process each line.

show-status-win <i>wid</i>	[Function]
kill-status-win <i>wid</i>	[Function]
status-win-entries <i>wid</i>	[Function]

These functions are used to manipulate status windows. **show-status-win** will expose a status window so that the user can see it. **kill-status-win** gets rid of the window permanently. **status-win-entries** is used to get the strings that have been written to a status window. This will return a list of strings.

make-comm-window <i>type</i>	[Function]
make-comm-status-window	[Function]
kill-comm-windows	[Function]

These are used for displaying windows containing data about communications between the SSM/PMAD interface and the rest of the breadboard. **make-comm-window** takes an argument *type* which may be either **:receiver** or **:transmitter** and makes the appropriate window for each. **make-comm-window** will display the communications log of the SSM/PMAD interface receiver or transmitter.

make-comm-status-window will make a window that displays the status of communications between the SSM/PMAD interface and the rest of the breadboard. This will show whether there is a connection or no connection to each of the other computers in the breadboard.

kill-comm-windows is a general cleanup function and will remove the different communication windows from the interface and reclaim their storage space.

screendump	[Function]
windowdump	[Function]

These two functions are used to perform a screendump and window dump of the interface respectively. The screendump function will start a background process that dumps the entire screen image of the solbourne to a file in the user's **knomad-archive** directory. The name of this file will start with **Screen** and be appended with the date and time of the dump. The **Windowdump** function will prompt the user for a particular window to dump and similarly write a file to the user's **knomad-archive** directory that starts with **Window** and is appended with the date and time of the dump.

legend	[Function]
summary-sub-menu-help	[Function]
utilization-sub-menu-help	[Function]
schedule-sub-menu-help	[Function]
summary-menu-help	[Function]
comm-sub-menu-help	[Function]
utils-menu-help	[Function]
knomad-menu-help	[Function]
init-sub-menu-help	[Function]
init-menu-help	[Function]
about-llps	[Function]
about-rpcs	[Function]
about-loads	[Function]
about-sensors	[Function]
about-cables	[Function]
about-cursor	[Function]
about-main-menu	[Function]
about-display-windows	[Function]

These functions are all used to display various help data to the user in the form of a status window (scrollable text window) for all but the legend function. The legend function displays a window that looks very similar to a status window but is not one. These functions are all accessible from the user interface and may be called programmatically.

3.4 Communications Functions

The communications functions of the SSM/PMAD interface come in two varieties. There are a few functions available for the purpose of manipulating connections between the computers of the breadboard and their states. The other variety of functions are for describing transactions that are sent along these communication channels. The functions for dealing with transactions are quite extensive, some are for describing transactions in general, while most are for the particular transactions unique to the SSM/PMAD breadboard.

Finally, there will be a third section on utility communication functions. All the functions related to communications are in the `comm` package.

3.4.1 Managing Communication Connections

In version 1.0 of the SSM/PMAD interface the user is somewhat limited on the amount of control available over a connection. To start with there are two important variables the user may access.

`*debug-comms*` [Variable]
`*process-table*` [Variable]

`*debug-comms*` may be set to a non-nil value by the user before starting the communications to enable debugging information to be kept. This debugging information is useful to the system developer that can access the various structures underlying the communications. For the general user, this will not be possible.

`*process-table*` is a table of information about the communications to each of the other computers of the breadboard. That is, `llp-a`, `llp-b`, `llp-c`, `llp-d`, `llp-e`, `llp-f`, `llp-g`, `llp-h`, and `symbolics`. The table includes the following information about each computer:

port The port is used to setup the underlying TCP socket port on the UNIX system for communicating with the specified computer.

hostname The hostname is the name of the computer in the UNIX system's host table that will be used to find the internet address of the computer.

stream The stream is used to record the current stream being used by the communications to communicate with this computer.

state The state is used to record what state this connection is in. The possible states are `connection` or `no-connection`.

queue The queue is used to record untransmitted messages to this computer that are waiting to be transmitted.

rcvr The rcvr is used to record the actual rcvr process of this communication channel that is used to receiver transactions.

xmtr The xmtr is similar to the rcvr but is used to transmit messages to this computer.

waiter The waiter is used for all the LLPs but not for the symbolics. It is used to record the process for accepting connections from an LLP. Perhaps *listener* would be a more appropriate name.

process	port	hostname
llp-a	14000	pc1
llp-b	14001	pc2
llp-c	14002	pc3
llp-d	14003	pc4
llp-e	14004	pc5
llp-f	14005	pc6
llp-g	14006	pc7
llp-h	14007	pc8
symbolics	9009	maestro

Table 1: Processes Ports and Hostnames

socket The socket is used to record what socket the UNIX systems has specified for us to use to communicate with this computer. We record this because once we are given a socket by the UNIX system we use that same socket every time we need to listen or accept a connection to one of the LLPs.

In version 1.0 of the system the port and hostname entries of the **process-table** are initialized as in table 1.

process-port <i>process</i>	[Function]
process-host <i>process</i>	[Function]
process-stream <i>process</i>	[Function]
process-state <i>process</i>	[Function]
process-queue <i>process</i>	[Function]
process-rcvr <i>process</i>	[Function]
process-xmtr <i>process</i>	[Function]
process-waiter <i>process</i>	[Function]
process-socket <i>process</i>	[Function]

These functions are used to access the entries on the **process-table**. *process* is one of the computers, e.g. llp-a, symbolics. All of these functions except for **process-port** and **process-host** are setf'able. However, it is recommended that the user does not do so unless absolutely certain of the setf's effects on the various communications processes.

Communications is a complex subject in its own right. Briefly, the Solbourne initiates a connection to the Symbolics, while the LLPs are responsible for initiating and maintaining

their connections to the Solbourne. Using TCP, to initiate a connection, one first gets a socket from the UNIX system then attempts to connect to the desired computer (using internet address and port number). This is done using the *open-connection* function for the Symbolics (described later). To receive and accept connections the protocol is a bit more complex. First a socket must be retrieved from the UNIX system, then a listen is set up on that socket related to a particular port. These ports are then connected to by the LLPs. When a connection is made by an LLP, the receiving software on the Solbourne may perform an accept. If the accept is successful a complete two-way connection has been made. The waiter function for the LLP is the primary mechanism for performing this sequence of functions and is also responsible for initializing the LLP when it first connects.

These concepts are very confusing to the amateur network software developer and also hard to explain to the general user. Since the functions available to the user for managing communications at this level are very limited, further confusion will be avoided. Suffice it to say that it works.

make-comm-processes

[Function]

kill-comm-processes

[Function]

make-comm-processes is used to start the communication processes and initiate a connection to the symbolics. This function is used by the SSM/PMAD interface to start the communications. *kill-comm-processes* is called when *stop-frames* is called and shuts down all the communications to the other computers in the breadboard.

open-connection host

[Function]

close-connection host

[Function]

These functions are used only while the normal communications processes are executing. *open-connection* may be used to open a connection to a computer where that connection is normally initiated by the Solbourne. In this case *host* may only be *symbolics*. *close-connection* may be used to close any connection. Here *host* may be any of *llp-a*, *llp-b*, ..., *symbolics*.

initialize-symbolics

[Function]

This function is used to initialize the Symbolics for autonomous operation. When autonomous mode is entered, this function is called. Therefore, if the user realizes that the connection to the symbolics has somehow been broken or not initiated, the user may first open the connection and then call this function to get things going again without starting over.

3.4.2 Transactions

Passing messages in the SSM/PMAD breadboard through the SSM/PMAD interface is done in the form of transactions. Transactions have a well-defined semantics for the system and are discussed here.

A transaction is a structure of four fields and defines the destination and source as well as the type of the data being communicated.

```
(defstruct transaction
  destination
  source
  type
  data)
```

The transaction, as well as all other types of data being communicated, is specified using a field-descriptor for each field of data:

```
(defstruct field-descriptor
  name
  length
  format
  repeat-next)
```

The field-descriptor is used to let the deblocking routines know how to take a transaction apart. The basic transaction has four fields and is defined to the deblocking routine using the following form:

```
(add-deblock 'transaction
  (list (make-field-descriptor :name "Destination"
                               :format 'symbol :length 1)
        (make-field-descriptor :name "Source" :format 'symbol :length 1)
        (make-field-descriptor :name "Message Type"
                               :format 'packed :length 1)
        (make-field-descriptor :name "Message Data" :length 'rest)))
```

All transactions are passed in the form of an ascii string with a newline at the end (ascii character 10). This string representation is a *marshalled* form of the data being passed. Marshalling the data for communication in the SSM/PMAD interface is done by blocking it and demarshalling is done by deblocking. Therefore, to specify a new data type to be passed between computers at least three items must be supplied: One, the structure of the

data, two, the deblock specification of the data, and three, the blocking specification of the data.⁵

The blocking specification takes the form of a function while the deblock specification is provided using the `add-deblock` function and field descriptors. There are three other important functions for each transaction that must also be provided by the transaction designer. The `transform-in` function is used to transform the incoming data into an alternate form that is more suitable to be used by the data processing functions. The `transform` function is used to transform data being transmitted as output to a form that the receiving computer is more interested in. For example, the LLPs transmit an RPC number as a two digit integer. The SSM/PMAD interface manipulates RPCs as symbols consisting of the three characters, the first is the identification letter of the LLP and the second two are the two digits that the LLP uses. The `transform-in` function processes this transformation. Finally, a third function, the `process` function is used to actually process the data.

The rest of this section will describe the various functions available to the designer of data communications and then the transactions used by the SSM/PMAD interface.

`add-deblock key descriptor` [Function]

`add-deblock` defines the description *descriptor* to be associated with the name *key*. The description should be a list of field descriptors.

`describe-deblock deblock-definition &key stream` [Function]

`describe-deblock` takes a definition of a deblock, either the name of the deblock as specified by `add-deblock` or the actual definition, and pretty prints it to *stream*. If *stream* is not provided it defaults to **standard-output**.

`field-descriptor name length format repeat-next` [Structure]

The `field-descriptor` is used to define the format of each field of a data type. The *name* is simply a description of the particular field and can be any data type. The *length* can be an integer or the symbol `rest` to specify how long the field is in the data stream. Alternatively *length* may also be the *key* of another deblock specification that means that this field is really a complete nested structure specified by a deblock. The *format* is used to specify how this field is formatted and how it should be read. The *repeat-next* slot defaults to `nil`. The *repeat-next* slot is used to specify that the next field descriptor should be repeated a number of times to produce a list of items. The exact number of times should be the result of the field descriptor that has the *repeat-next* specified as non-`nil`.

⁵However, one may consider if the data is only input or output. If it is only input, then only the blocking specification does not need to be provided and vice versa. It is good practice to provide all the structure as well as both blocking and deblock specifications.

The best way to use the field descriptors are to examine the examples of the communications of the SSM/PMAD interface in the source listings.

The format of data specified in a field descriptor may currently be of the following forms:

normal	[<i>Deblock Format</i>]
symbol	[<i>Deblock Format</i>]
packed	[<i>Deblock Format</i>]
packed79	[<i>Deblock Format</i>]
llp-integer	[<i>Deblock Format</i>]
numeric	[<i>Deblock Format</i>]

The **normal** format will simply read the number of characters of the string specified by the *length* slot of the field descriptor. **normal** reads a string. **symbol** is like **normal** except that the string read by **normal** will be interned into the **rule-system** package and returned as the symbol. **packed** specifies that the current character of the input string is in a packed format and needs to be unpacked. The **packed** format allows integers in the range 0 to 79 to be represented as one ascii character. The *length* slot for a **packed** format should always be 1. The **packed79** format is like **packed** except that it will use two ascii characters to specify a base 79 number. The *length* slot for **packed79** should be 2. The **numeric** format reads the straight ascii representation of an integer from the input string based upon the *length* slot. The **llp-integer** format is like the **numeric** format except that the bytes are reversed.

The rest of this section on transactions describes the SSM/PMAD interface transactions.

transaction-table	[<i>Variable</i>]
transaction-address	[<i>Variable</i>]

The ***transaction-table*** is used to define the transactions of the SSM/PMAD interface. Each entry consists of three items: The name of the transaction, its type, and its priority. This table is used by the various functions defined next.

The ***transaction-address*** table is used to specify the destination and source of transactions. These addresses are all single character codes for the various software components on the computers of the breadboard.

<code>transaction-id</code> <i>code</i>	[<i>Function</i>]
<code>transaction-blocking-function</code> <i>type</i>	[<i>Function</i>]
<code>transaction-transform-function</code> <i>type</i>	[<i>Function</i>]
<code>transaction-transform-in-function</code> <i>type</i>	[<i>Function</i>]
<code>transaction-priority</code> <i>type</i>	[<i>Function</i>]
<code>transaction-process-function</code> <i>type</i>	[<i>Function</i>]
<code>transaction-address</code> <i>code</i>	[<i>Function</i>]

These functions are used to retrieve various aspects about transactions from either the name of a transaction or its code (where *type* and *code* will be bound to these). `transaction-id` will return the name of the transaction if the code is given and the code if the name is given. The `transaction-blocking-function`, `transaction-transform-function`, `transaction-transform-in-function`, and `transaction-process-function` functions return the function associated with the transaction type specified (either by name or code). `transaction-priority-function` will return the priority of the transaction. `transaction-address` will return the long name of the address if given the one character representation and vice versa.

The transaction table and addresses are pictured in tables 2 and 3 respectively. The transaction definitions follow.

SYNC-TIME

```
(defstruct sync-time
```

```
  now-month
  now-day
  now-year
  now-hour
  now-minute
  now-second
  som-month
  som-day
  som-year
  som-hour
  som-minute
  som-second)
```

```
(add-deblock 'sync-time
```

```
  (list (make-field-descriptor :name "Now Month"
```

<i>Name</i>	<i>Code</i>	<i>Priority</i>
Symbolics → Solbourne		
sync-time	1	high
event-list	2	normal
priority-list	3	normal
contingency-events	8	high
ready?	11	normal
source-power-change	14	high
Solbourne → Symbolics		
switch-to-redundant	6	high
load-shed	7	high
out-of-service	9	high
utilization	10	low
ready!	12	normal
initialized	13	normal
contingency-start	15	high
contingency-end	16	high
Solbourne → LLP		
llp-event-list	30	normal
llp-priority-list	31	normal
llp-time-list	32	high
llp-contingency-events	33	normal
switch-control	34	normal
switch-conversion-constants	35	normal
sensor-conversion-constants	36	normal
initialize	37	normal
llp-query	38	high
LLP → Solbourne		
switch-status	40	normal
sensor-status	41	normal
temp-sensor-status	42	normal
switch-performance	43	normal
sensor-performance	44	normal
switch-conversion-values	45	normal
sensor-conversion-values	46	normal
q-status	49	high

Table 2: *transaction-table*

<i>Computer</i>	<i>Code</i>
LLP-A	A
LLP-B	B
LLP-C	C
LLP-D	D
LLP-E	E
LLP-F	F
LLP-G	G
LLP-H	H
Symbolics	S
FELES	S
MAESTRO	S
LPLMS	S
FRAMES	W
UNIX-Box	W
Workstation	W
Solbourne	W
FRAMES	P
FRAMES	X

Table 3: *transaction-address*

```
                                :format 'numeric :length 2)
(make-field-descriptor :name "Now Day" :format 'numeric :length 2)
(make-field-descriptor :name "Now Year" :format 'numeric :length 2)
(make-field-descriptor :name "Now Hour" :format 'numeric :length 2)
(make-field-descriptor :name "Now Minute"
                        :format 'numeric :length 2)
(make-field-descriptor :name "Now Second"
                        :format 'numeric :length 2)
(make-field-descriptor :name "SOM Month"
                        :format 'numeric :length 2)
(make-field-descriptor :name "SOM Day" :format 'numeric :length 2)
(make-field-descriptor :name "SOM Year" :format 'numeric :length 2)
(make-field-descriptor :name "SOM Hour" :format 'numeric :length 2)
(make-field-descriptor :name "SOM Minute"
                        :format 'numeric :length 2)
(make-field-descriptor :name "SOM Second"
                        :format 'numeric :length 2)))
```

EVENT-LIST

```
(defstruct event-list
  effective-time
  number-of-events
  events)
```

```
(defstruct event
  time
  component-id
  type
  max-power
  permission-to-test
  redundancy
  switch-to-redundant
  max-current
  min-current
  min-power)
```

```
(add-deblock 'event
  (list (make-field-descriptor :name "Time of Event"
                                :format 'numeric :length 6)
```



```
(make-field-descriptor :name "Component" :format 'symbol :length 3)
(make-field-descriptor :name "Event" :format 'symbol :length 1)
(make-field-descriptor :name "Max Power"
                        :format 'numeric :length 5)
(make-field-descriptor :name "Permission to Test"
                        :format 'symbol :length 1)
(make-field-descriptor :name "Redundancy"
                        :format 'symbol :length 1)
(make-field-descriptor :name "Switch to Redundant"
                        :format 'symbol :length 1)
(make-field-descriptor :name "Max Current"
                        :format 'numeric :length 3)
(make-field-descriptor :name "Min Current"
                        :format 'numeric :length 3)
(make-field-descriptor :name "Min Power"
                        :format 'numeric :length 5)))
```

```
(add-deblock 'event-list
  (list (make-field-descriptor :name "Effective Time"
                              :format 'numeric :length 6)
        (make-field-descriptor :name "Number of Events"
                              :format 'packed79 :length 2 :repeat-next t)
        (make-field-descriptor :name "Events" :length 'event)))
```

LOAD-PRIORITY-LIST

```
(defstruct load-priority-list
  effective-time
  number-of-components
  components)
```

```
(add-deblock 'priority-list
  (list (make-field-descriptor :name "Effective Time"
                              :format 'numeric :length 6)
        (make-field-descriptor :name "Number of Components"
                              :format 'packed79 :length 2 :repeat-next t)
        (make-field-descriptor :name "Components"
                              :format 'symbol :length 3)))
```

SWITCH-TO-REDUNDANT

```
(defstruct switch-to-redundant
  number-of-entries
  specs)

(defstruct redundant-switch
  component-id
  time)

(add-deblock 'switch-to-redundant
  (list (make-field-descriptor :name "Number of Redundant Switches"
                              :format 'packed79 :length 2 :repeat-next t)
        (make-field-descriptor :name "Redundant Switch"
                              :length 'redundant-switch)))

(add-deblock 'redundant-switch
  (list (make-field-descriptor :name "Component"
                              :format 'symbol :length 3)
        (make-field-descriptor :name "Time of Switch"
                              :format 'numeric :length 6)))
```

LOAD-SHED

```
(defstruct load-shed
  number-of-entries
  specs)

(defstruct shed-load
  component-id
  time)

(add-deblock 'load-shed
  (list (make-field-descriptor :name "Number of Load Sheds"
                              :format 'packed79 :length 2 :repeat-next t)
        (make-field-descriptor :name "Load Sheds"
                              :length 'shed-load)))

(add-deblock 'shed-load
```

```
(list (make-field-descriptor :name "Component"
                             :format 'symbol :length 3)
      (make-field-descriptor :name "Time of Shed"
                             :format 'numeric :length 6)))
```

CONTINGENCY

```
(defstruct contingency
  effective-time
  number-of-entries
  events)
```

```
(add-deblock 'contingency-events
  (list (make-field-descriptor :name "Effective Time"
                              :format 'numeric :length 6)
        (make-field-descriptor :name "Number of Events"
                              :format 'packed79 :length 2 :repeat-next t)
        (make-field-descriptor :name "Events" :length 'event))))
```

OUT-OF-SERVICE

```
(defstruct out-of-service
  number-of-entries
  specs)
```

```
(defstruct service-outage
  component-id
  start
  end)
```

```
(add-deblock 'out-of-service
  (list (make-field-descriptor :name "Number of Out of Services"
                              :format 'packed79 :length 2 :repeat-next t)
        (make-field-descriptor :name "Service Outages"
                              :length 'service-outage))))
```

```
(add-deblock 'service-outage
  (list (make-field-descriptor :name "Component" :format 'symbol :length 3)
```

```
(make-field-descriptor :name "Begin Time of Outage"
                        :format 'numeric :length 6)
(make-field-descriptor :name "End Time of Outage"
                        :format 'numeric :length 6)))
```

UTILIZATION

```
(defstruct utilization
  bus-a-start
  bus-a-end
  number-of-bus-a-entries
  bus-a-entries
  bus-b-start
  bus-b-end
  number-of-bus-b-entries
  bus-b-entries)
```

```
(defstruct power-spec
  component-id
  power)
```

```
(add-deblock 'utilization
  (list (make-field-descriptor :name "Bus A Begin Time"
                                :format 'numeric :length 6)
        (make-field-descriptor :name "Bus A End Time"
                                :format 'numeric :length 6)
        (make-field-descriptor :name "Number of Power Utilizations"
                                :format 'packed79 :length 2 :repeat-next t)
        (make-field-descriptor :name "Power Utilization"
                                :length 'power-utilization)
        (make-field-descriptor :name "Bus B Begin Time"
                                :format 'numeric :length 6)
        (make-field-descriptor :name "Bus B End Time"
                                :format 'numeric :length 6)
        (make-field-descriptor :name "Number of Power Utilizations"
                                :format 'packed79 :length 2 :repeat-next t)
        (make-field-descriptor :name "Power Utilization"
                                :length 'power-utilization))))
```

```
(add-deblock 'power-utilization
(list (make-field-descriptor :name "Component" :format 'symbol :length 3)
      (make-field-descriptor :name "Utilization"
                             :format 'numeric :length 5)))
```

READY?

```
(add-deblock 'ready?
(list (make-field-descriptor :name "Ready?" :format 'symbol :length 1)))
```

READY!

```
(add-deblock 'ready!
(list (make-field-descriptor :name "Ready!" :format 'symbol :length 1)))
```

INITIALIZED

```
(add-deblock 'initialized
(list (make-field-descriptor :name "Initialized"
                             :format 'symbol :length 1)))
```

SOURCE-POWER-CHANGE

```
(defstruct source-power-change
  start
  end
  power)
```

```
(add-deblock 'source-power-change
(list (make-field-descriptor :name "Effective Start Time"
                              :format 'numeric :length 6)
      (make-field-descriptor :name "Effective End Time"
                              :format 'numeric :length 6)
      (make-field-descriptor :name "Source Power"
                              :format 'numeric :length 5)))
```

CONTINGENCY-START

```
(add-deblock 'contingency-start
(list (make-field-descriptor :name "Contingency Start Time"
                              :format 'numeric :length 6)))
```

CONTINGENCY-END

```
(add-deblock 'contingency-end
  (list (make-field-descriptor :name "Contingency End Time"
                               :format 'numeric :length 6)))
```

LLP-EVENT-LIST

```
(defstruct llp-event-list
  effective-time
  number-of-events
  events)
```

```
(defstruct llp-event
  time
  component-id
  event
  type
  redundancy
  switch-to-redundant
  max-current
  min-current)
```

```
(add-deblock 'llp-event
  (list (make-field-descriptor :name "Time of Event"
                               :format 'numeric :length 6)
        (make-field-descriptor :name "Component" :format 'symbol :length 3)
        (make-field-descriptor :name "Event" :format 'symbol :length 1)
        (make-field-descriptor :name "Event Type"
                               :format 'symbol :length 1)
        (make-field-descriptor :name "Redundancy"
                               :format 'symbol :length 1)
        (make-field-descriptor :name "Switch to Redundant"
                               :format 'symbol :length 1)
        (make-field-descriptor :name "Max Current"
                               :format 'numeric :length 3)
        (make-field-descriptor :name "Min Current"
                               :format 'numeric :length 3)))
```

```
(add-deblock 'llp-event-list
  (list (make-field-descriptor :name "Effective Time"
```

```
                                :format 'numeric :length 6)
(make-field-descriptor :name "Number of Events"
                        :format 'packed79 :length 2 :repeat-next t)
(make-field-descriptor :name "Events" :length 'llp-event)))
```

LLP-PRIORITY-LIST

```
(defstruct llp-priority-list
  effective-time
  number-of-components
  components)

(add-deblock 'llp-priority-list
  (list (make-field-descriptor :name "Effective Time"
                                :format 'numeric :length 6)
        (make-field-descriptor :name "Number of Components"
                                :format 'packed79 :length 2 :repeat-next t)
        (make-field-descriptor :name "Components"
                                :format 'symbol :length 3))))
```

LLP-TIME-LIST

```
(defstruct llp-time-list
  now-month
  now-day
  now-year
  now-hour
  now-minute
  now-second
  som-month
  som-day
  som-year
  som-hour
  som-minute
  som-second)

(add-deblock 'llp-time-list
  (list (make-field-descriptor :name "Now Month"
                                :format 'numeric :length 2)
        (make-field-descriptor :name "Now Day" :format 'numeric :length 2)
```

```
(make-field-descriptor :name "Now Year" :format 'numeric :length 2)
(make-field-descriptor :name "Now Hour" :format 'numeric :length 2)
(make-field-descriptor :name "Now Minute"
                        :format 'numeric :length 2)
(make-field-descriptor :name "Now Second"
                        :format 'numeric :length 2)
(make-field-descriptor :name "SOM Month"
                        :format 'numeric :length 2)
(make-field-descriptor :name "SOM Day" :format 'numeric :length 2)
(make-field-descriptor :name "SOM Year" :format 'numeric :length 2)
(make-field-descriptor :name "SOM Hour" :format 'numeric :length 2)
(make-field-descriptor :name "SOM Minute"
                        :format 'numeric :length 2)
(make-field-descriptor :name "SOM Second"
                        :format 'numeric :length 2)))
```

LLP-CONTINGENCY-EVENTS

```
(defstruct llp-contingency-events
  effective-time
  number-of-events
  events)
```

```
(add-deblock 'llp-contingency-events
  (list (make-field-descriptor :name "Effective Time"
                              :format 'numeric :length 6)
        (make-field-descriptor :name "Number of Events"
                              :format 'packed79 :length 2 :repeat-next t)
        (make-field-descriptor :name "Events" :length 'llp-event)))
```

SWITCH-CONTROL

```
(defstruct switch-control
  effective-time
  number-of-events
  events)
```

```
(add-deblock 'switch-control
  (list (make-field-descriptor :name "Effective Time"
                              :format 'numeric :length 6)
```



```
(make-field-descriptor :name "Number of Events"
                        :format 'packed79 :length 2 :repeat-next t)
(make-field-descriptor :name "Events" :length 'llp-event)))
```

SWITCH-CONVERSION-CONSTANTS

```
(defstruct switch-conversion-constants
  number-of-constants
  switch-constants)

(defstruct switch-constant
  component-id
  slope
  intercept)

(add-deblock 'switch-conversion-constants
  (list (make-field-descriptor :name "Number of Constants"
                              :format 'llp-integer :length 4
                              :repeat-next t)
        (make-field-descriptor :name "Switch Constants"
                              :length 'switch-constant))))
```

SENSOR-CONVERSION-CONSTANTS

```
(defstruct sensor-conversion-constants
  number-of-constants
  sensor-constants)

(defstruct sensor-constant
  component-id
  i-slope
  i-intercept
  v-slope
  v-intercept
  p-slope
  p-intercept
  t-slope
  t-intercept)

(add-deblock 'sensor-constant
```

```
(list (make-field-descriptor :name "Component"
                             :format 'llp-integer :length 4)
      (make-field-descriptor :name "Current Slope"
                             :format 'llp-integer :length 4)
      (make-field-descriptor :name "Current Intercept" :
                             format 'llp-integer :length 4)
      (make-field-descriptor :name "Voltage Slope"
                             :format 'llp-integer :length 4)
      (make-field-descriptor :name "Voltage Intercept"
                             :format 'llp-integer :length 4)
      (make-field-descriptor :name "Power Slope"
                             :format 'llp-integer :length 4)
      (make-field-descriptor :name "Power Intercept"
                             :format 'llp-integer :length 4)
      (make-field-descriptor :name "Temperature Slope"
                             :format 'llp-integer :length 4)
      (make-field-descriptor :name "Temperature Intercept"
                             :format 'llp-integer :length 4)))
```

LLP-QUERY

```
(defstruct llp-query
  type
  llp)
```

```
(add-deblock 'llp-query
  (list (make-field-descriptor :name "Query Type"
                              :format 'symbol :length 1)))
```

SWITCH-STATUS

```
(defstruct switch-status
  switch-num
  anomalous
  llp-flags
  number-switch-records
  switch-records)
```

```
(defstruct switch-record
  component-id)
```

status
hold-type
status-word
current
trip-tag)

```
(add-deblock 'switch-status
  (list (make-field-descriptor :name "Switch Number"
                               :format 'llp-integer :length 4)
        (make-field-descriptor :name "Anomalous" :format 'symbol :length 1)
        (make-field-descriptor :name "LLP Flags"
                               :format 'llp-integer :length 4)
        (make-field-descriptor :name "Number of Switch Records"
                               :format 'llp-integer :length 4
                               :repeat-next t)
        (make-field-descriptor :name "Switch Records"
                               :length 'switch-record)))
```

```
(add-deblock 'switch-record
  (list (make-field-descriptor :name "Component"
                               :format 'llp-integer :length 4)
        (make-field-descriptor :name "Position" :format 'symbol :length 1)
        (make-field-descriptor :name "Hold Type" :format 'symbol :length 1)
        (make-field-descriptor :name "Status Word"
                               :format 'llp-integer :length 4)
        (make-field-descriptor :name "Current"
                               :format 'llp-integer :length 4)
        (make-field-descriptor :name "Trip Tag"
                               :format 'llp-integer :length 4)))
```

SENSOR-STATUS

```
(defstruct sensor-status
  number-of-sensor-records
  sensor-records)
```

```
(defstruct sensor-record
  component-id
  current
  voltage)
```

power
state)

```
(add-deblock 'sensor-status
  (list (make-field-descriptor :name "Number of Sensor Records"
                                :format 'llp-integer :length 4
                                :repeat-next t)
        (make-field-descriptor :name "Sensor Records"
                                :length 'sensor-record)))
```

```
(add-deblock 'sensor-record
  (list (make-field-descriptor :name "Component"
                                :format 'llp-integer :length 4)
        (make-field-descriptor :name "Current"
                                :format 'llp-integer :length 4)
        (make-field-descriptor :name "Voltage"
                                :format 'llp-integer :length 4)
        (make-field-descriptor :name "Power"
                                :format 'llp-integer :length 4)
        (make-field-descriptor :name "State"
                                :format 'llp-integer :length 4)))
```

TEMP-SENSOR-STATUS

```
(defstruct temp-sensor-status
  number-of-records
  temp-sensor-records)
```

```
(defstruct temp-sensor-record
  component-id
  temperature)
```

```
(add-deblock 'temp-sensor-status
  (list (make-field-descriptor :name "Number of Records"
                                :format 'llp-integer :length 4
                                :repeat-next t)
        (make-field-descriptor :name "Temperature Sensor Records"
                                :length 'temp-sensor-record)))
(add-deblock 'temp-sensor-record
  (list (make-field-descriptor :name "Component"
```

```
                                :format 'llp-integer :length 4)
(make-field-descriptor :name "Temperature"
                        :format 'llp-integer :length 4)))
```

SWITCH-PERFORMANCE

```
(defstruct switch-performance
  number-of-switches
  switch-amp-records)
```

```
(defstruct switch-amp-record
  component-id
  start-time
  end-time
  avg-current
  max-current
  min-current
  max-time
  min-time)
```

```
(add-deblock 'switch-performance
  (list (make-field-descriptor :name "Number of Switches"
                                :format 'llp-integer :length 4
                                :repeat-next t)
        (make-field-descriptor :name "Switch Amperage Records"
                                :length 'switch-amp-record)))
```

```
(add-deblock 'switch-amp-record
  (list (make-field-descriptor :name "Component"
                                :format 'llp-integer :length 4)
        (make-field-descriptor :name "Start Time"
                                :format 'llp-integer :length 4)
        (make-field-descriptor :name "End Time"
                                :format 'llp-integer :length 4)
        (make-field-descriptor :name "Average Current"
                                :format 'llp-integer :length 4)
        (make-field-descriptor :name "Maximum Current"
                                :format 'llp-integer :length 4)
        (make-field-descriptor :name "Minimum Current"
                                :format 'llp-integer :length 4)))
```

```
(make-field-descriptor :name "Maximum Time"
                        :format 'llp-integer :length 4)
(make-field-descriptor :name "Minimum Time"
                        :format 'llp-integer :length 4)))
```

SENSOR-PERFORMANCE

```
(defstruct sensor-performance
  start-time
  end-time
  number-of-records
  sensor-amp-records)
```

```
(defstruct sensor-amp-record
  avg-voltage
  max-voltage
  min-voltage
  avg-current
  max-current
  min-current
  avg-power
  max-power
  min-power
  energy-consumed)
```

```
(add-deblock 'sensor-performance
  (list (make-field-descriptor :name "Start Time"
                              :format 'llp-integer :length 4)
        (make-field-descriptor :name "End Time"
                              :format 'llp-integer :length 4)
        (make-field-descriptor :name "Number of Records"
                              :format 'llp-integer :length 4
                              :repeat-next t)
        (make-field-descriptor :name "Sensor Amperage Records"
                              :length 'sensor-amp-record)))
```

```
(add-deblock 'sensor-amp-record
  (list (make-field-descriptor :name "Average Voltage"
                              :format 'llp-integer :length 4)
        (make-field-descriptor :name "Maximum Voltage"
```

```
                                :format 'llp-integer :length 4)
(make-field-descriptor :name "Minimum Voltage"
                        :format 'llp-integer :length 4)
(make-field-descriptor :name "Average Current"
                        :format 'llp-integer :length 4)
(make-field-descriptor :name "Maximum Current"
                        :format 'llp-integer :length 4)
(make-field-descriptor :name "Minimum Current"
                        :format 'llp-integer :length 4)
(make-field-descriptor :name "Average Power"
                        :format 'llp-integer :length 4)
(make-field-descriptor :name "Maximum Power"
                        :format 'llp-integer :length 4)
(make-field-descriptor :name "Minimum Power"
                        :format 'llp-integer :length 4)
(make-field-descriptor :name "Energy Consumed"
                        :format 'llp-integer :length 4)))
```

SWITCH-CONVERSION-VALUES

```
(defstruct switch-conversion-values
  number-of-constants
  switch-constants)
```

```
(add-deblock 'switch-conversion-values
  (list (make-field-descriptor :name "Number of Constants"
                                :format 'llp-integer :length 4
                                :repeat-next t)
        (make-field-descriptor :name "Switch Constants"
                                :length 'switch-constant))))
```

```
(add-deblock 'switch-constant
  (list (make-field-descriptor :name "Component"
                                :length 4 :format 'llp-integer)
        (make-field-descriptor :name "Slope"
                                :format 'llp-integer :length 4)
        (make-field-descriptor :name "Intercept"
                                :format 'llp-integer :length 4))))
```

SENSOR-CONVERSION-VALUES

```
(defstruct sensor-conversion-values
  number-of-constants
  sensor-constants)
```

```
(add-deblock 'sensor-conversion-values
  (list (make-field-descriptor :name "Number of Constants"
                               :format 'llp-integer :length 4
                               :repeat-next t)
        (make-field-descriptor :name "Sensor Constants"
                               :length 'sensor-constant))))
```

SWITCH-SENSOR-CONFIG

```
(defstruct switch-sensor-config
  sensors?
  number-of-switches
  switch-configs)
```

```
(defstruct switch-config
  component-id
  type
  state)
```

```
(add-deblock 'switch-sensor-config
  (list (make-field-descriptor :name "Sensors Available?"
                               :format 'symbol :length 1)
        (make-field-descriptor :name "Number of Switches"
                               :format 'llp-integer :length 4
                               :repeat-next t)
        (make-field-descriptor :name "Switch Configurations"
                               :length 'switch-config))))
```

```
(add-deblock 'switch-config
  (list (make-field-descriptor :name "Component"
                               :format 'llp-integer :length 4)
        (make-field-descriptor :name "Switch Type"
                               :format 'symbol :length 1)
        (make-field-descriptor :name "Switch State"
                               :format 'symbol :length 1))))
```


Q-STATUS

```
(defstruct q-status
  status)
```

```
(add-deblock 'q-status
  (list (make-field-descriptor :name "Status" :format 'symbol :length 1)))
```

3.4.3 Utility Communication Functions

There are a number of supporting functions defined to support using the communications and sending transactions. There are also some functions for working with transactions and their data that has been received. These are defined here.

<code>switch-shed? <i>status-word</i></code>	[Macro]
<code>switched-to-redundant? <i>status-word</i></code>	[Macro]
<code>tripped? <i>status-word</i></code>	[Macro]
<code>get-trip-type <i>status-word</i></code>	[Macro]
<code>bit-info <i>status-word</i></code>	[Function]

These are used to get information out of the bit-defined four-byte integer that is sent from the LLP to the Solbourne with each switch record in the switch-status transaction. `switch-shed?`, `switched-to-redundant?`, and `tripped?` all return a `t` or `nil` value. `get-trip-type` returns a trip type (or `nil`), which can be one of: `fast-trip`, `over-current`, `under-voltage`, and `ground-fault`. `bit-info` returns a list of all the bits that are set in the *status-word*, if any.

<code>switch-switch-to-redundant <i>switch</i></code>	[Function]
---	------------

This function is used to update the database of the domain to specify that the given *switch* has been switched to redundant.

<code>gather-data-and-start-diagnosis</code>	[Function]
<code>get-snapshot</code>	[Function]

These functions are used when a fault has been detected in the power system. `get-snapshot` performs a complex function of communication with the LLPs to find out what trips the switches may have. `get-snapshot` returns a *symptom-set*. `get-snapshot` is used by the fault diagnosis expert system when isolating the fault. `gather-data-and-start-diagnosis` is called when a fault is first detected by an LLP. It calls `get-snapshot` to find out the current

symptoms and asserts the symptoms to the database so that the expert system may perform a fault diagnosis.

<code>send-load-sheds components &optional time</code>	[Function]
<code>send-redundants components &optional time</code>	[Function]
<code>send-out-of-service-switches switches &optional time</code>	[Function]
<code>send-q-status-msg llp</code>	[Function]
<code>send-next-data-msg llp</code>	[Function]
<code>send-next-sensor-msg llp</code>	[Function]
<code>send-contingency-start &optional time</code>	[Function]
<code>send-contingency-end &optional time</code>	[Function]
<code>send-switch-control-events events &optional time &key :queue</code>	[Function]
<code>continuous-info-on switch-number</code>	[Function]
<code>continuous-info-off switch-number</code>	[Function]

These functions are used to send various messages either to the LLPs or to the Symbolics. `send-load-sheds`, `send-redundants`, and `send-out-of-service-switches` are used to communicate to FELES and MAESTRO the new states of these particular switches. In these three functions *components*, and *switches* are all switch names (symbols such as a03). In all these functions where *time* is an optional parameter, it specifies when the particular event happened, such as a switch being taken out of service. If *time* is not specified it defaults to the value of the global variable **last-update-time** which is always updated in autonomous mode.

`send-q-status-msg` is used to tell the LLPs that they should respond as soon as they are in a quiescent state. This function is used by the `get-snapshot` routine.

`send-next-data-msg` and `send-next-sensor-msg` are used to ask the LLP given to send a switch-status or sensor-status message respectively.

`send-contingency-start` and `send-contingency-end` are used to let FELES and MAESTRO know that a contingency is being processed.

`send-switch-control-events` is used to command switches on and off. Each event of *events* should be a proper *llp-event* as specified in the transactions.

`continuous-switch-info-on` and `continuous-switch-info-off` are functions for commanding the LLP to send data about a switch every pass through its loop. These are primarily for the user interface so that the user may observe what is happening to a switch. These functions must be used with some care as version 1.0 of the system is not optimized. Having

an LLP continuously send switch data every pass through its loop may have a tendency to degrade the performance of the software on the Solbourne.

3.5 Utility Functions

The SSM/PMAD interface provides a number of utility functions for general support of the rest of the system. These functions are for clock support, events, and queues.

clock [Variable]

units-per-second [Variable]

The clock in the SSM/PMAD interface is bound to the global variable ***clock***. The Solbourne system clock is not used because to change the time one would have to be root, which is not good practice for general software development and applications.

The ***units-per-second*** variable is provided for the purpose of adjusting the speed of the clock either faster or slower. ***units-per-second*** defaults to 1. If this is changed to 2 then the clock will run twice as fast as the wall clock. It is recommended that this variable not be changed. The rest of the SSM/PMAD interface does not use **clock-sleep** as often as it should to properly schedule system functions.

make-clock [Function]

start-clock *cl* [Method]

kill-clock *cl* [Method]

get-time *cl* [Method]

set-time *cl time* [Method]

clock-sleep *seconds* [Function]

These functions are used for manipulating the clock. **make-clock** makes the clock and returns it. **start-clock** and **kill-clock** are methods and must be passed the clock as an argument. **make-clock** does not start the clock, **start-clock** must be used to do this. **get-time** and **set-time** are used to get the time of the clock and set the time respectively. For setting the time, *time* should be in universal time format. Finally, **clock-sleep** is used as an alternative to the LISP **sleep** function to take into account the speed of the clock.

create-event &optional *name* [Function]

await-event *event* &optional *timeout* [Function]

notify-event *event* [Function]

An event is used by a process to wait for something. Generally, an event can be thought

of as a situation. To wait for an event a process calls **await-event**. To wake up from an event some other process must call **notify-event** on the event that the sleeping process is waiting for. Of course the event must be first created with **create-event**.

When a process waits for an event, **await-event** will add a reason to the process-arrest-reasons of the process. This effectively takes the process off of the scheduling queue and puts it to sleep. It is woken up when **notify-event** called with the same event removes the reason from the process-arrest-reasons.

await-event may also take an optional *timeout* argument. This argument is the number of SECONDS that the process is willing to wait for a notification. If the notification does not come in that time, it will be woken up regardless. If the timeout is not given, the process may potentially wait forever.

An event may be notified before a process is awaiting it. If this happens the signal is stored in the event. The first process that awaits the event will then be immediately woken up. Multiple notifications to an event do NOT stack. They will be treated as one notification.

If there is more than one process waiting for an event, they will all be woken up if the event is signaled. This is probably the way one would do events in OS programs as well. If it turns out that all the processes are waiting on the same resource and only one can have it then you have a case where you would want to use Dijkstra's locking algorithm. There is a difference here between events and locks that I am trying to distinguish. Events are used to signal things in general. Locks are used to obtain exclusive access rights over something. Events may cause locking operations, while locking mechanisms may use events internally in some manner.

This manner of event handling is VERY similar to the Xerox notion of events.

Some more thoughts as a result of scanning "An Introduction to Operating Systems" by Harvey M. Deitel (Addison-Wesley, 1984).

Events are generally used for synchronization. This implementation generalizes the basic event mechanism in two ways; one, so multiple processes can wait on the same event, and two, a waiting process can have a timeout — (is this similar to some of Ada's mechanisms for process synchronization?). Another generalization of the event mechanism is to allow a process to wait on a boolean combination of events. To do this one would want to define events as bits in a word. As events are signaled, blocked processes (waiting on events) are anded (logically) with the signaled event to see if they are woken up...

<code>create-queue &optional name</code>	[Function]
<code>init-queue queue</code>	[Function]
<code>init-queues</code>	[Function]
<code>add-entry queue item</code>	[Function]
<code>remove-entry queue</code>	[Function]
<code>get-queue-entries queue</code>	[Function]

The queue functions provide general support for the queue data structure. In this implementation a queue operation is always done in constant time by utilizing the pointer capabilities of LISP.

`create-queue` is used to make a new queue. `init-queue` will initialize a queue. This is good to do if the state of the queue is unknown. `init-queues` will initialize every queue that has been defined. `add-entry` and `remove-entry` will add and remove entries from a queue. `get-queue-entries` returns a list of all the entries on a queue, allowing the user to perform other functions on queue items if desired.

4 FRAMES Technical Reference

The FRAMES system is a large, complex set of programs for fault diagnosis, isolation, and recovery in the world of a space station module-like power system. It consists of a number of traditional expert systems, a number of conventional algorithmic control processes, as well as a large amount of advanced programming techniques. The programs making up FRAMES reside on everything from PC clones to UNIX boxes to specialized LISP machines. The FRAMES system is an advanced research and development platform for the purpose of designing a robust fault detection, isolation, and recovery mechanism that could be applicable to Space Station Freedom.

This section of the report briefly overviews the architecture of FRAMES, describes the modular rule organization of the multiple faults expert system part of FRAMES, and then provides excruciating detail on the FRAMES knowledge base.

4.1 The FRAMES Architecture

The FRAMES system is partitioned into three major divisions based upon the response time needed at the different partitions. The three partitions are the distributed lower level processors for controlling the hardware, the fault isolation and diagnosis expert systems, and the scheduling system.

The switch hardware is controlled by the lower level processes which reside on PC clones. The algorithmic processes at this level control the operation of turning switches on and off as well as monitoring power levels and performing limit checking. The lower level processes detect fault conditions which include a switch physically tripping off due to an over current or under voltage situation in the hardware. These *fault symptoms* are communicated to the fault isolation and diagnosis expert systems. Additionally, scheduled operations may no longer be performed on the tripped switches.

The response time of the lower level processors is necessarily fast. Typically, limit checking operations to shut a load off if it is using too much power, for example, are done in within a one second period. The speed of the lower level processes also has implications on later fault isolation. It is quite possible that if an I^2t short is occurring in the hardware at a level of approximately 120% of the switch's rating, that it could take the switch up to five seconds to trip. The lower level processor will shut the switch off much sooner than this.

The third partition, the scheduling system, is not required to be nearly as responsive. Its role is to create a schedule for operating the switches in advance and to maintain that schedule during contingencies in the power system. In the present system, schedules are shipped to the lower level processors in thirty minute blocks. This allows for a semi-graceful degradation of the overall system performance if the scheduler becomes inoperable for some reason. When a fault has been diagnosed in the power system and a set of switches has been determined unusable, the scheduler is expected to reschedule its activities in a reasonable

amount of time. The scheduler has been partitioned at the highest level and is expected to perform in a period of minutes.

The second partition is the fault isolation and diagnosis part of FRAMES. This part consists of a number of traditional expert systems for diagnosing different types of problems in the power hardware as well as maintaining other knowledge intensive states such as the load priority list. Currently three expert systems are defined to exist at this level: the Load Priority List Management System (LPLMS), the fault diagnosis expert system, and the soft fault expert system.

The fault isolation and diagnosis expert system requires many supporting functions in addition to the rules that make it up. Embedding knowledge intensive applications into real world complex systems require many parts for a successful system (see [Rieb] for one way to deal with this). In the FRAMES system these additional functions include detecting and monitoring the hardware (done by the lower level processors); communication algorithms for communicating with the distributed processors; algorithmic processes for logging data, updating database values, and the like; and user interface functions to make the system useful.

A result of this modular organization of functions in both inter and intraprocessors is that the expert systems for fault isolation and diagnosis do not need to be executing for a person to use the system. Another way to look at it is that operating the system in an autonomous fashion requires the reasoning processes as embedded in the expert systems, while operating it manually does not.

4.2 Multiple Faults in SSM/PMAD

The problem of multiple faults in SSM/PMAD can be divided into two cases:

Case 1 Faults that occur within Δ time of one another.

Case 2 Faults that occur at least Δ time from one another.

Where Δ is defined as: *The amount of time it takes for a fault to be initially detected and subsequently diagnosed.* Faults that occur at least Δ time from one another were already handled in the first generation of FRAMES. Faults that occur within Δ time of one another are the focus of this report.

Suppose first that multiple faults have occurred in the power system during the detection of the faults. By the time the power system has reached a quiescent state, the lower level processors will report a set of symptoms indicative of more than one fault. The fault isolation software is then tasked with determining how these collected symptoms might indicate multiple faults.

There are three cases that may be identified. The multiple faults may occur on the same bus, they may occur in the same hierarchy, and they may occur on completely independent

buses. For multiple faults that occur in the same hierarchy it is possible that one of the faults could be masked (by a bad current sensor, for example) and appear to be multiple faults on the same bus.

To adequately deal with multiple faults the set of symptoms that the LLPs report are first analyzed and organized into clusters. A cluster of symptoms is a set such that each symptom in the set either occurred on the same bus or occurred below another symptom. This leaves two cases of multiple faults for the expert system to deal with. Each cluster may be dealt with independently.

Given a cluster of symptoms, the first thing that is checked is if all the top symptoms (those symptoms of the set that are all at the highest level – all on the same bus therefore) are under voltage. If this is the case it is possible that there may be no power to the bus. To check this, the top sensor of the bus as well as various sensors above the tripped switches are looked at to see if they have nominal voltage or less than nominal voltage.

If all the top symptoms are either fast trip or over current and they all have loads hooked up that are related to the same activity, it is possible that the particular activity may be involved in the trips. If all the top symptoms are fast trip, not related by an activity and do not have any switches below them, then it is possible that one of the switches had a short below it and the other switches may have fast tripped due to energy storage (but unlikely).

Finally, if none of the above cases apply, each of the top symptoms is diagnosed as an independent fault indication. Each top symptom will be either fast trip or over current (the under voltages were diagnosed earlier). The particular top switch and the switches below it may then be tested and diagnosed as an independent fault. Now, if the fault is found somewhere below the top switch (due to a masked fault), there may have been other faults in that hierarchy. If there were, the highest (in the topology of switches) of these other faults, below the top symptom yet across from the switch finally diagnosed as the position of the fault, may also be diagnosed as independent faults.

An added complication is that the isolation and diagnosis phase is part of the Δ time. This includes commanding switches on and off in an effort to repeat the symptoms. If another fault occurs during this testing, the data collection algorithms must be smart enough to incorporate any new symptoms correctly into the existing symptoms.

See [Riea] for more details about multiple faults in the SSM/PMAD breadboard.

4.3 The FRAMES Knowledge Base

The FRAMES knowledge base is defined using the KNOMAD-SSM/PMAD system (described in section 5). The knowledge base is first listed here.

```
00
00 The FRAMES knowledge base
00
00 This knowledge base is for defining the rules and data necessary
```



```
00 for implementing the expert system for power diagnosis.
00 It is currently set up as three rule groups.
00 The control rule group controls invocation of the other two.
00 It watches for symptoms from the power system and calls the
00 hard fault rule group to perform diagnosis on any symptoms.
00 When a diagnosis is determined, the diagnosis rule group is
00 called for printing the diagnosis out and setting up any necessary out of
00 service information.
00
```

KB : MF-frames

```
00
00 We start by initializing the domain. This includes LOTS of
00 information to define the model.
00
```

@DOMAIN : /usr/local/knomad/domains/frames.domain

00 Load the rule groups

```
FILE : /usr/local/knomad/knowledge-bases/mf-control.rg
FILE : /usr/local/knomad/knowledge-bases/multiple-fault.rg
FILE : /usr/local/knomad/knowledge-bases/mf-diagnosis.rg
FILE : /usr/local/knomad/knowledge-bases/soft-fault.rg
```

Domain-Knowledge :

constants :

```
t ; true ; false ; nil ;
started ; done ;
multiple-hard-fault ; mf-diagnosis ;
on ; off ; yes ; no ; y ; n ;
over-current ; under-voltage ; fast-trip ;
00 Diagnosis slots
:name ; :top-symp ; :slot1 ; :slot2 ;
00 soft-fault constants
:analyzed ; :unanalyzed ;
00
00 the diagnosis constants
00
no-power-to-bus ;
broken-cable-between-sensor-above-and-u-v-switches ;
broke-output-cable-of-switch-above ;
broke-input-cable-of-switch-above ;
break-in-cable-above-switch-above-and-bad-u-v-sensor-switch-above ;
no-permission-to-test-possible-backrush ;
unexpected-to-many-retrips-possible-backrush ;
no-retrips-on-flips-possible-backrush ;
```

```

found-possible-backrush ;
unexpected-retrip-possible-backrush ;
unexpected-symptoms-during-open ;
unexpected-too-many-symptoms-flip-top ;
retrip-on-flip ;
unexpected-retrip-during-flip ;
not-found-no-levels ;
unexpected-trips-during-close-top ;
not-found-cant-test-further ;
unexpected-to-many-tops-after-flips ;
unexpected-different-top-after-flips ;
not-found-all-tested ;
possible-found ;
unexpected-different-trip-during-closes ;
unexpected-new-trips-during-closes .
facts :
empty = ( ) ;
sf-nodes = ( a-node1 a-node2 a-node3 a-node4 a-node5 a-node6 a-node7
             a-node8 h-node1 h-node2 h-node3 h-node4 h-node5 h-node6
             h-node7 h-node8 b-node1 b-node2 c-node1 c-node2 d-node1
             d-node2 e-node1 e-node2 f-node1 f-node2 g-node1 g-node2 ) ;
sf-result = ( ) .

%%
%% Let's start the control rule group
%%
begin : mf-control

end-kb

```

In a typical stand-alone environment the knowledge base for a knowledge agent is loaded. This involves defining the domain and defining the rules of the knowledge agent. When applying a knowledge agent to an embedded system, such as SSM/PMAD, how the various pieces of the knowledge agent get loaded may be changed. In the FRAMES knowledge agent we load the domain explicitly from another location. When we need to run the SSM/PMAD breadboard in autonomous mode the rest of the knowledge agent gets loaded, that is the knowledge base definition given above.

The organization of a knowledge agent consists of a number of parts. Generally the first part will define the domain. In the above listing, that part is commented out. Then the rule groups are defined. These can be made a part of the same file as the knowledge agent or can be split into different files for easier maintenance. Some initial domain knowledge, in the form of constants, facts, and frames is then defined. Finally, those rule groups that should start executing are specified.

The next three subsections list the FRAMES domain and the expert systems making up the FRAMES knowledge agent.

4.3.1 The FRAMES Domain

The FRAMES domain is currently specified in the form of a LISP file. This involves defining the frames making up the domain and the actual objects of the domain.

;;; The Domain Objects

```
(frame :name power-domain
      :slots ((top-symptoms :value #'top-symptoms)
              (cluster-symptoms :value #'cluster-symptoms)
              (flip-switches :value #'flip-switches)
              (close-switches :value #'close-switches)
              (open-switches :value #'open-switches)
              (open-relevant-switches :value #'open-relevant-switches)
              (flip-switch :value #'flip-switch)
              (close-switch :value #'close-switch)
              (reclose-switches :value #'reclose-switches)
              (new-diagnosable-switches :value #'new-diagnosable-switches)
              (out-of-service :value #'out-of-service)
              (send-out-of-services :value #'send-out-of-services)
              (end-contingency :value #'end-contingency)
              (make-diagnosis :value #'make-diagnosis1)
              (sum-values :value #'sum-values)
              (loose-< :value #'loose-<)
              (loose-> :value #'loose->)
              (loose-= :value #'loose-=)
              (loose->= :value #'loose->=)
              (kludge-switch :value #'kludge-switch)
              (kludge-fault :value #'kludge-fault)
              (write :value #'domain-write)
              (diagnosis-window :value t)))
```

```
(frame :name diagnosis
      :slots ((name :value nil)
              (top-symp :value nil)
              (slot1 :value nil)
              (slot2 :value nil)))
```

```
(frame :name node
      :slots ((upper-switch :value nil)
              (lower-switches :value nil)
              (upper-sensor :value nil)
              (lower-sensors :value nil)
              (upper-node :value nil)
              (lower-nodes :value nil)
              (analyzed :value :unanalyzed)))
```

```
(frame :name symptom-set
```

```
:slots ((symptoms)))

(frame :name symptom
:slots ((switch)
(fault)))

(frame :name event
:slots ((command)
(component-id)
(max-power)
(redundancy)
(switch-to-redundant)
(max-current)
(min-current)
(min-power)
(permission-to-test)
(time)))

(frame :name source
:slots ((name)
(voltage)
(current)
(available :value t)
(cable-out :value nil)))

(frame :name load
:slots ((name)
(cable-in :value nil)
(left-bottom)
(active-region)
(powered :value nil)
(llp)
(voltage)
(current)
(restartable :value nil)))

(frame :name llp
:slots ((name)
(window)
(left-bottom)
(width-height)
(available :value t)
(sic-a :value :available)
(sic-b :value :available)
(contained-switches-bus-a)
(contained-switches-bus-b)
(contained-sensors)))
```

```
(frame :name sensor
  :slots ((name)
    (current)
    (voltage)
    (power)
    (temperature)
    (available :value t)
    (i-slope)
    (i-intercept)
    (v-slope)
    (v-intercept)
    (t-slope)
    (t-intercept)
    (performance-data-stack :value nil)
    (latest-performance-current-avg :value 0)
    (active-region)
    (upper-node)
    (lower-node)))
```

```
(frame :name switch
  :slots ((name)
    (type)
    (left-bottom)
    (active-region)
    (state :value :open)
    (available :value t)
    (tripped :value nil)
    (powered :value nil)
    (current)
    (voltage)
    (switches-below :value nil)
    (switch-above :value nil)
    (siblings)
    (event)
    (mf-indication :value :false)
    (status-word) ;from llp
    (current) ;from llp
    (trip-tag :value nil) ;from llp
    (slope)
    (intercept)
    (corresponding-sensors :value nil)
    (sensor-above :value nil)
    (top-sensor)
    (current-trippable)
    (current-rating)
    (fast-trip-percent)
    (over-current-percent)
    (under-voltage-trippable))
```

```
(under-voltage-value)
(llp)
(redundant-switch :value nil)
(power-rating)
(cable-in :value nil)
(cable-out :value nil)
(performance-data-stack :value nil)
(latest-performance-current-avg :value 0)
(upper-node)
(lower-node)))

(frame :name cable
      :slots ((name)
              (polygons)
              (powered :value nil)
              (in)
              (out)))

(defvar *voltage* 120) ; DC voltage

(fcreate-instance 'symptom-set 'symptom-set1)
(fcreate-instance 'symptom 'top-symptom)

;;; The Domain Definition

;;; create the switches first

;;;a01
(dolist (switch '(a01))
  (fcreate-instance 'switch switch)
  (assert! '(frame ,switch type :value rbi))
  (assert! '(frame ,switch name :value ,switch))
  (assert! '(frame ,switch current-trippable :value false))
  (assert! '(frame ,switch under-voltage-trippable :value false))
  (assert! '(frame ,switch current-rating ,(/ 15000 *voltage*)))
  (assert! '(frame ,switch fast-trip-percent :value 1.75))
  (assert! '(frame ,switch over-current-percent :value 1.20))
  (assert! '(frame ,switch llp :value llp-a))
  (assert! '(frame ,switch power-rating :value 15000))
  (assert! '(frame ,switch upper-node :value a-node1))
  (assert! '(frame ,switch lower-node :value a-node2))
  (assert! '(frame ,switch cable-in :value llp-a-cable-0))
  (assert! '(frame ,switch cable-out :value llp-a-cable-1)))

;;;a02 a03 a04 a05 a06 a07
(dolist (switch '(a02 a03 a04 a05 a06 a07))
  (fcreate-instance 'switch switch)
  (assert! '(frame ,switch type :value 3k-rpc)))
```

```
(assert! '(frame ,switch name :value ,switch))
(assert! '(frame ,switch current-trippable :value true))
(assert! '(frame ,switch current-rating :value ,(/ 3000 *voltage*)))
(assert! '(frame ,switch fast-trip-percent :value 1.75))
(assert! '(frame ,switch over-current-percent :value 1.20))
(assert! '(frame ,switch under-voltage-trippable :value true))
(assert! '(frame ,switch under-voltage-value :value 60))
(assert! '(frame ,switch llp :value llp-a))
(assert! '(frame ,switch power-rating :value 3000))
(assert! '(frame ,switch upper-node :value a-node2))
(assert! '(frame ,switch cable-in :value llp-a-cable-1)))

(assert! '(frame a02 lower-node :value a-node3))
(assert! '(frame a03 lower-node :value a-node4))
(assert! '(frame a04 lower-node :value a-node5))
(assert! '(frame a05 lower-node :value a-node6))
(assert! '(frame a06 lower-node :value a-node7))
(assert! '(frame a07 lower-node :value a-node8))

(assert! '(frame a02 cable-out :value llp-a-cable-2))
(assert! '(frame a03 cable-out :value llp-a-cable-3))
(assert! '(frame a04 cable-out :value llp-a-cable-4))
(assert! '(frame a05 cable-out :value llp-a-cable-5))
(assert! '(frame a06 cable-out :value llp-a-cable-6))
(assert! '(frame a07 cable-out :value llp-a-cable-7))

;;;h01
(dolist (switch '(h01))
  (fcreate-instance 'switch switch)
  (assert! '(frame ,switch type :value rbi))
  (assert! '(frame ,switch name :value ,switch))
  (assert! '(frame ,switch current-trippable :value false))
  (assert! '(frame ,switch under-voltage-trippable :value false))
  (assert! '(frame ,switch current-rating :value ,(/ 15000 *voltage*)))
  (assert! '(frame ,switch fast-trip-percent :value 1.75))
  (assert! '(frame ,switch over-current-percent :value 1.20))
  (assert! '(frame ,switch llp :value llp-h))
  (assert! '(frame ,switch power-rating :value 15000))
  (assert! '(frame ,switch upper-node :value h-node1))
  (assert! '(frame ,switch lower-node :value h-node2))
  (assert! '(frame ,switch cable-in :value llp-h-cable-0))
  (assert! '(frame ,switch cable-out :value llp-h-cable-1)))

;;;h02 h03 h04 h05 h06 h07
(dolist (switch '(h02 h03 h04 h05 h06 h07))
  (fcreate-instance 'switch switch)
  (assert! '(frame ,switch type :value 3k-rpc))
  (assert! '(frame ,switch name :value ,switch)))
```

```
(assert! '(frame ,switch current-trippable :value true))
(assert! '(frame ,switch current-rating :value ,(/ 3000 *voltage*)))
(assert! '(frame ,switch fast-trip-percent :value 175))
(assert! '(frame ,switch over-current-percent :value 120))
(assert! '(frame ,switch under-voltage-trippable :value true))
(assert! '(frame ,switch under-voltage-value :value 60))
(assert! '(frame ,switch llp :value llp-h))
(assert! '(frame ,switch power-rating :value 3000))
(assert! '(frame ,switch upper-node :value h-node2))
(assert! '(frame ,switch cable-in :value llp-h-cable-1)))

(assert! '(frame h02 lower-node :value h-node3))
(assert! '(frame h03 lower-node :value h-node4))
(assert! '(frame h04 lower-node :value h-node5))
(assert! '(frame h05 lower-node :value h-node6))
(assert! '(frame h06 lower-node :value h-node7))
(assert! '(frame h07 lower-node :value h-node8))

(assert! '(frame h02 cable-out :value llp-h-cable-2))
(assert! '(frame h03 cable-out :value llp-h-cable-3))
(assert! '(frame h04 cable-out :value llp-h-cable-4))
(assert! '(frame h05 cable-out :value llp-h-cable-5))
(assert! '(frame h06 cable-out :value llp-h-cable-6))
(assert! '(frame h07 cable-out :value llp-h-cable-7))

;;;load center b switches
(dolist (switch '(b00 b01 b02 b03 b04 b05 b06 b07 b08 b14 b15 b16 b17 b18
b19 b20 b21 b22))
  (fcreate-instance 'switch switch)
  (assert! '(frame ,switch type :value 1k-rpc))
  (assert! '(frame ,switch name :value ,switch))
  (if (< (read-from-string (subseq (format nil "~a" switch) 1)) 10)
    (assert! '(frame ,switch cable-in :value llp-a-cable-2))
    (assert! '(frame ,switch cable-in :value llp-h-cable-2)))
  (assert! '(frame ,switch current-trippable :value true))
  (assert! '(frame ,switch current-rating :value ,(/ 1000 *voltage*)))
  (assert! '(frame ,switch fast-trip-percent :value 175))
  (assert! '(frame ,switch over-current-percent :value 120))
  (assert! '(frame ,switch under-voltage-trippable :value true))
  (assert! '(frame ,switch under-voltage-value :value 60))
  (assert! '(frame ,switch llp :value llp-b))
  (assert! '(frame ,switch power-rating :value 1000)))

;;;load center c switches
(dolist (switch '(c00 c01 c02 c03 c04 c05 c06 c07 c08 c14 c15 c16 c17 c18
c19 c20 c21 c22))
  (fcreate-instance 'switch switch)
  (assert! '(frame ,switch type :value 1k-rpc))
```



```
(assert! '(frame ,switch name :value ,switch))
(if (< (read-from-string (subseq (format nil "~a" switch) 1)) 10)
  (assert! '(frame ,switch cable-in :value llp-a-cable-3))
  (assert! '(frame ,switch cable-in :value llp-h-cable-3)))
(assert! '(frame ,switch current-trippable :value true))
(assert! '(frame ,switch current-rating :value ,(/ 1000 *voltage*)))
(assert! '(frame ,switch fast-trip-percent :value 175))
(assert! '(frame ,switch over-current-percent :value 120))
(assert! '(frame ,switch under-voltage-trippable :value true))
(assert! '(frame ,switch under-voltage-value :value 60))
(assert! '(frame ,switch llp :value llp-c))
(assert! '(frame ,switch power-rating :value 1000)))

;;;load center d switches
(dolist (switch '(d00 d01 d02 d03 d04 d05 d06 d07 d08 d14 d15 d16 d17 d18
  d19 d20 d21 d22))
  (fcreate-instance 'switch switch)
  (assert! '(frame ,switch type :value 1k-rpc))
  (assert! '(frame ,switch name :value ,switch))
  (if (< (read-from-string (subseq (format nil "~a" switch) 1)) 10)
    (assert! '(frame ,switch cable-in :value llp-a-cable-4))
    (assert! '(frame ,switch cable-in :value llp-h-cable-4)))
  (assert! '(frame ,switch current-trippable :value true))
  (assert! '(frame ,switch current-rating :value ,(/ 1000 *voltage*)))
  (assert! '(frame ,switch fast-trip-percent :value 175))
  (assert! '(frame ,switch over-current-percent :value 120))
  (assert! '(frame ,switch under-voltage-trippable :value true))
  (assert! '(frame ,switch under-voltage-value :value 60))
  (assert! '(frame ,switch llp :value llp-d))
  (assert! '(frame ,switch power-rating :value 1000)))

;;;load center e switches
(dolist (switch '(e00 e01 e02 e03 e04 e05 e06 e07 e08 e14 e15 e16 e17 e18
  e19 e20 e21 e22))
  (fcreate-instance 'switch switch)
  (assert! '(frame ,switch type :value 1k-rpc))
  (assert! '(frame ,switch name :value ,switch))
  (if (< (read-from-string (subseq (format nil "~a" switch) 1)) 10)
    (assert! '(frame ,switch cable-in :value llp-a-cable-5))
    (assert! '(frame ,switch cable-in :value llp-h-cable-5)))
  (assert! '(frame ,switch current-trippable :value true))
  (assert! '(frame ,switch current-rating :value ,(/ 1000 *voltage*)))
  (assert! '(frame ,switch fast-trip-percent :value 175))
  (assert! '(frame ,switch over-current-percent :value 120))
  (assert! '(frame ,switch under-voltage-trippable :value true))
  (assert! '(frame ,switch under-voltage-value :value 60))
  (assert! '(frame ,switch llp :value llp-e))
  (assert! '(frame ,switch power-rating :value 1000)))
```

```
;;;load center f switches
(dolist (switch '(f00 f01 f02 f03 f04 f05 f06 f07 f08 f14 f15 f16 f17 f18
  f19 f20 f21 f22))
  (fcreate-instance 'switch switch)
  (assert! '(frame ,switch type :value 1k-rpc))
  (assert! '(frame ,switch name :value ,switch))
  (if (< (read-from-string (subseq (format nil "~a" switch) 1)) 10)
    (assert! '(frame ,switch cable-in :value llp-a-cable-6))
    (assert! '(frame ,switch cable-in :value llp-h-cable-6)))
  (assert! '(frame ,switch current-trippable :value true))
  (assert! '(frame ,switch current-rating :value ,(/ 1000 *voltage*)))
  (assert! '(frame ,switch fast-trip-percent :value 175))
  (assert! '(frame ,switch over-current-percent :value 120))
  (assert! '(frame ,switch under-voltage-trippable :value true))
  (assert! '(frame ,switch under-voltage-value :value 60))
  (assert! '(frame ,switch llp :value llp-f))
  (assert! '(frame ,switch power-rating :value 1000)))

;;;load center g switches
(dolist (switch '(g00 g01 g02 g03 g04 g05 g06 g07 g08 g14 g15 g16 g17 g18
  g19 g20 g21 g22))
  (fcreate-instance 'switch switch)
  (assert! '(frame ,switch type :value 1k-rpc))
  (assert! '(frame ,switch name :value ,switch))
  (if (< (read-from-string (subseq (format nil "~a" switch) 1)) 10)
    (assert! '(frame ,switch cable-in :value llp-a-cable-7))
    (assert! '(frame ,switch cable-in :value llp-h-cable-7)))
  (assert! '(frame ,switch current-trippable :value true))
  (assert! '(frame ,switch current-rating :value ,(/ 1000 *voltage*)))
  (assert! '(frame ,switch fast-trip-percent :value 175))
  (assert! '(frame ,switch over-current-percent :value 120))
  (assert! '(frame ,switch under-voltage-trippable :value true))
  (assert! '(frame ,switch under-voltage-value :value 60))
  (assert! '(frame ,switch llp :value llp-g))
  (assert! '(frame ,switch power-rating :value 1000)))

(assert! '(frame b00 cable-out :value llp-b-cable-0))
(assert! '(frame b01 cable-out :value llp-b-cable-1))
(assert! '(frame b02 cable-out :value llp-b-cable-2))
(assert! '(frame b03 cable-out :value llp-b-cable-3))
(assert! '(frame b04 cable-out :value llp-b-cable-4))
(assert! '(frame b05 cable-out :value llp-b-cable-5))
(assert! '(frame b06 cable-out :value llp-b-cable-6))
(assert! '(frame b07 cable-out :value llp-b-cable-7))
(assert! '(frame b08 cable-out :value llp-b-cable-8))
(assert! '(frame b14 cable-out :value llp-b-cable-14))
(assert! '(frame b15 cable-out :value llp-b-cable-15))
```

```
(assert! '(frame b16 cable-out :value llp-b-cable-16))  
(assert! '(frame b17 cable-out :value llp-b-cable-17))  
(assert! '(frame b18 cable-out :value llp-b-cable-18))  
(assert! '(frame b19 cable-out :value llp-b-cable-19))  
(assert! '(frame b20 cable-out :value llp-b-cable-20))  
(assert! '(frame b21 cable-out :value llp-b-cable-21))  
(assert! '(frame b22 cable-out :value llp-b-cable-22))
```

```
(assert! '(frame c00 cable-out :value llp-c-cable-0))  
(assert! '(frame c01 cable-out :value llp-c-cable-1))  
(assert! '(frame c02 cable-out :value llp-c-cable-2))  
(assert! '(frame c03 cable-out :value llp-c-cable-3))  
(assert! '(frame c04 cable-out :value llp-c-cable-4))  
(assert! '(frame c05 cable-out :value llp-c-cable-5))  
(assert! '(frame c06 cable-out :value llp-c-cable-6))  
(assert! '(frame c07 cable-out :value llp-c-cable-7))  
(assert! '(frame c08 cable-out :value llp-c-cable-8))  
(assert! '(frame c14 cable-out :value llp-c-cable-14))  
(assert! '(frame c15 cable-out :value llp-c-cable-15))  
(assert! '(frame c16 cable-out :value llp-c-cable-16))  
(assert! '(frame c17 cable-out :value llp-c-cable-17))  
(assert! '(frame c18 cable-out :value llp-c-cable-18))  
(assert! '(frame c19 cable-out :value llp-c-cable-19))  
(assert! '(frame c20 cable-out :value llp-c-cable-20))  
(assert! '(frame c21 cable-out :value llp-c-cable-21))  
(assert! '(frame c22 cable-out :value llp-c-cable-22))
```

```
(assert! '(frame d00 cable-out :value llp-d-cable-0))  
(assert! '(frame d01 cable-out :value llp-d-cable-1))  
(assert! '(frame d02 cable-out :value llp-d-cable-2))  
(assert! '(frame d03 cable-out :value llp-d-cable-3))  
(assert! '(frame d04 cable-out :value llp-d-cable-4))  
(assert! '(frame d05 cable-out :value llp-d-cable-5))  
(assert! '(frame d06 cable-out :value llp-d-cable-6))  
(assert! '(frame d07 cable-out :value llp-d-cable-7))  
(assert! '(frame d08 cable-out :value llp-d-cable-8))  
(assert! '(frame d14 cable-out :value llp-d-cable-14))  
(assert! '(frame d15 cable-out :value llp-d-cable-15))  
(assert! '(frame d16 cable-out :value llp-d-cable-16))  
(assert! '(frame d17 cable-out :value llp-d-cable-17))  
(assert! '(frame d18 cable-out :value llp-d-cable-18))  
(assert! '(frame d19 cable-out :value llp-d-cable-19))  
(assert! '(frame d20 cable-out :value llp-d-cable-20))  
(assert! '(frame d21 cable-out :value llp-d-cable-21))  
(assert! '(frame d22 cable-out :value llp-d-cable-22))
```

```
(assert! '(frame e00 cable-out :value llp-e-cable-0))  
(assert! '(frame e01 cable-out :value llp-e-cable-1))
```

```
(assert! '(frame e02 cable-out :value llp-e-cable-2))
(assert! '(frame e03 cable-out :value llp-e-cable-3))
(assert! '(frame e04 cable-out :value llp-e-cable-4))
(assert! '(frame e05 cable-out :value llp-e-cable-5))
(assert! '(frame e06 cable-out :value llp-e-cable-6))
(assert! '(frame e07 cable-out :value llp-e-cable-7))
(assert! '(frame e08 cable-out :value llp-e-cable-8))
(assert! '(frame e14 cable-out :value llp-e-cable-14))
(assert! '(frame e15 cable-out :value llp-e-cable-15))
(assert! '(frame e16 cable-out :value llp-e-cable-16))
(assert! '(frame e17 cable-out :value llp-e-cable-17))
(assert! '(frame e18 cable-out :value llp-e-cable-18))
(assert! '(frame e19 cable-out :value llp-e-cable-19))
(assert! '(frame e20 cable-out :value llp-e-cable-20))
(assert! '(frame e21 cable-out :value llp-e-cable-21))
(assert! '(frame e22 cable-out :value llp-e-cable-22))
```

```
(assert! '(frame f00 cable-out :value llp-f-cable-0))
(assert! '(frame f01 cable-out :value llp-f-cable-1))
(assert! '(frame f02 cable-out :value llp-f-cable-2))
(assert! '(frame f03 cable-out :value llp-f-cable-3))
(assert! '(frame f04 cable-out :value llp-f-cable-4))
(assert! '(frame f05 cable-out :value llp-f-cable-5))
(assert! '(frame f06 cable-out :value llp-f-cable-6))
(assert! '(frame f07 cable-out :value llp-f-cable-7))
(assert! '(frame f08 cable-out :value llp-f-cable-8))
(assert! '(frame f14 cable-out :value llp-f-cable-14))
(assert! '(frame f15 cable-out :value llp-f-cable-15))
(assert! '(frame f16 cable-out :value llp-f-cable-16))
(assert! '(frame f17 cable-out :value llp-f-cable-17))
(assert! '(frame f18 cable-out :value llp-f-cable-18))
(assert! '(frame f19 cable-out :value llp-f-cable-19))
(assert! '(frame f20 cable-out :value llp-f-cable-20))
(assert! '(frame f21 cable-out :value llp-f-cable-21))
(assert! '(frame f22 cable-out :value llp-f-cable-22))
```

```
(assert! '(frame g00 cable-out :value llp-g-cable-0))
(assert! '(frame g01 cable-out :value llp-g-cable-1))
(assert! '(frame g02 cable-out :value llp-g-cable-2))
(assert! '(frame g03 cable-out :value llp-g-cable-3))
(assert! '(frame g04 cable-out :value llp-g-cable-4))
(assert! '(frame g05 cable-out :value llp-g-cable-5))
(assert! '(frame g06 cable-out :value llp-g-cable-6))
(assert! '(frame g07 cable-out :value llp-g-cable-7))
(assert! '(frame g08 cable-out :value llp-g-cable-8))
(assert! '(frame g14 cable-out :value llp-g-cable-14))
(assert! '(frame g15 cable-out :value llp-g-cable-15))
(assert! '(frame g16 cable-out :value llp-g-cable-16))
```

```
(assert! '(frame g17 cable-out :value llp-g-cable-17))
(assert! '(frame g18 cable-out :value llp-g-cable-18))
(assert! '(frame g19 cable-out :value llp-g-cable-19))
(assert! '(frame g20 cable-out :value llp-g-cable-20))
(assert! '(frame g21 cable-out :value llp-g-cable-21))
(assert! '(frame g22 cable-out :value llp-g-cable-22))

;;; create the non-existent switches

(dolist (switch '(a00 a08 a09 a10 a11 a12 a13 a14 a15 a16 a17 a18 a19 a20
a21 a22 a23 a24 a25 a26 a27
h00 h08 h09 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20
h21 h22 h23 h24 h25 h26 h27
b09 b10 b11 b12 b13 b23 b24 b25 b26 b27
c09 c10 c11 c12 c13 c23 c24 c25 c26 c27
d09 d10 d11 d12 d13 d23 d24 d25 d26 d27
e09 e10 e11 e12 e13 e23 e24 e25 e26 e27
f09 f10 f11 f12 f13 f23 f24 f25 f26 f27
g09 g10 g11 g12 g13 g23 g24 g25 g26 g27))
(fcreate-instance 'switch switch)
(assert! '(frame ,switch name :value ,switch)))

;;; sensors

(dolist (sensor '(am0 am1 am2 am3 am4 am5 am6 am7))
(fcreate-instance 'sensor sensor)
(assert! '(frame ,sensor name :value ,sensor)))

(dolist (sensor '(hm0 hm1 hm2 hm3 hm4 hm5 hm6 hm7))
(fcreate-instance 'sensor sensor)
(assert! '(frame ,sensor name :value ,sensor)))

(dolist (sensor '(bm0 bm1 cm0 cm1 dm0 dm1 em0 em1 fm0 fm1 gm0 gm1))
(fcreate-instance 'sensor sensor)
(assert! '(frame ,sensor name :value ,sensor)))

;;; make non-existent sensors

(dolist (sensor '(am8 am9 am10 am11 am12 am13 am14 am15
hm8 hm9 hm10 hm11 hm12 hm13 hm14 hm15
bm2 bm3 bm4 bm5 bm6 bm7 bm8 bm9 bm10 bm11 bm12 bm13 bm14 bm15
cm2 cm3 cm4 cm5 cm6 cm7 cm8 cm9 cm10 cm11 cm12 cm13 cm14 cm15
dm2 dm3 dm4 dm5 dm6 dm7 dm8 dm9 dm10 dm11 dm12 dm13 dm14 dm15
em2 em3 em4 em5 em6 em7 em8 em9 em10 em11 em12 em13 em14 em15
fm2 fm3 fm4 fm5 fm6 fm7 fm8 fm9 fm10 fm11 fm12 fm13 fm14 fm15
gm2 gm3 gm4 gm5 gm6 gm7 gm8 gm9 gm10 gm11 gm12 gm13 gm14 gm15
))
(fcreate-instance 'sensor sensor))
```

```
(assert! '(frame ,sensor name :value ,sensor)))

;;; start hooking things together

;;;siblings

(assert! '(frame a02 siblings :value (a03 a04 a05 a06 a07)))
(assert! '(frame a03 siblings :value (a02 a04 a05 a06 a07)))
(assert! '(frame a04 siblings :value (a02 a03 a05 a06 a07)))
(assert! '(frame a05 siblings :value (a02 a03 a04 a06 a07)))
(assert! '(frame a06 siblings :value (a02 a03 a04 a05 a07)))
(assert! '(frame a07 siblings :value (a02 a03 a04 a05 a06)))

(assert! '(frame h02 siblings :value (h03 h04 h05 h06 h07)))
(assert! '(frame h03 siblings :value (h02 h04 h05 h06 h07)))
(assert! '(frame h04 siblings :value (h02 h03 h05 h06 h07)))
(assert! '(frame h05 siblings :value (h02 h03 h04 h06 h07)))
(assert! '(frame h06 siblings :value (h02 h03 h04 h05 h06)))
(assert! '(frame h07 siblings :value (h02 h03 h04 h05 h07)))

(assert! '(frame b00 siblings :value (b01 b02 b03 b04 b05 b06 b07 b08)))
(assert! '(frame b01 siblings :value (b00 b02 b03 b04 b05 b06 b07 b08)))
(assert! '(frame b02 siblings :value (b00 b01 b03 b04 b05 b06 b07 b08)))
(assert! '(frame b03 siblings :value (b00 b01 b02 b04 b05 b06 b07 b08)))
(assert! '(frame b04 siblings :value (b00 b01 b02 b03 b05 b06 b07 b08)))
(assert! '(frame b05 siblings :value (b00 b01 b02 b03 b04 b06 b07 b08)))
(assert! '(frame b06 siblings :value (b00 b01 b02 b03 b04 b05 b07 b08)))
(assert! '(frame b07 siblings :value (b00 b01 b02 b03 b04 b05 b06 b08)))
(assert! '(frame b08 siblings :value (b00 b01 b02 b03 b04 b05 b06 b07)))

(assert! '(frame b14 siblings :value (b15 b16 b17 b18 b19 b20 b21 b22)))
(assert! '(frame b15 siblings :value (b14 b16 b17 b18 b19 b20 b21 b22)))
(assert! '(frame b16 siblings :value (b14 b15 b17 b18 b19 b20 b21 b22)))
(assert! '(frame b17 siblings :value (b14 b15 b16 b18 b19 b20 b21 b22)))
(assert! '(frame b18 siblings :value (b14 b15 b16 b17 b19 b20 b21 b22)))
(assert! '(frame b19 siblings :value (b14 b15 b16 b17 b18 b20 b21 b22)))
(assert! '(frame b20 siblings :value (b14 b15 b16 b17 b18 b19 b21 b22)))
(assert! '(frame b21 siblings :value (b14 b15 b16 b17 b18 b19 b20 b22)))
(assert! '(frame b22 siblings :value (b14 b15 b16 b17 b18 b19 b20 b21)))

(assert! '(frame c00 siblings :value (c01 c02 c03 c04 c05 c06 c07 c08)))
(assert! '(frame c01 siblings :value (c00 c02 c03 c04 c05 c06 c07 c08)))
(assert! '(frame c02 siblings :value (c00 c01 c03 c04 c05 c06 c07 c08)))
(assert! '(frame c03 siblings :value (c00 c01 c02 c04 c05 c06 c07 c08)))
(assert! '(frame c04 siblings :value (c00 c01 c02 c03 c05 c06 c07 c08)))
(assert! '(frame c05 siblings :value (c00 c01 c02 c03 c04 c06 c07 c08)))
(assert! '(frame c06 siblings :value (c00 c01 c02 c03 c04 c05 c07 c08)))
(assert! '(frame c07 siblings :value (c00 c01 c02 c03 c04 c05 c06 c08)))
```

(assert! '(frame c08 siblings :value (c00 c01 c02 c03 c04 c05 c06 c07)))

(assert! '(frame c14 siblings :value (c15 c16 c17 c18 c19 c20 c21 c22)))
(assert! '(frame c15 siblings :value (c14 c16 c17 c18 c19 c20 c21 c22)))
(assert! '(frame c16 siblings :value (c14 c15 c17 c18 c19 c20 c21 c22)))
(assert! '(frame c17 siblings :value (c14 c15 c16 c18 c19 c20 c21 c22)))
(assert! '(frame c18 siblings :value (c14 c15 c16 c17 c19 c20 c21 c22)))
(assert! '(frame c19 siblings :value (c14 c15 c16 c17 c18 c20 c21 c22)))
(assert! '(frame c20 siblings :value (c14 c15 c16 c17 c18 c19 c21 c22)))
(assert! '(frame c21 siblings :value (c14 c15 c16 c17 c18 c19 c20 c22)))
(assert! '(frame c22 siblings :value (c14 c15 c16 c17 c18 c19 c20 c21)))

(assert! '(frame d00 siblings :value (d01 d02 d03 d04 d05 d06 d07 d08)))
(assert! '(frame d01 siblings :value (d00 d02 d03 d04 d05 d06 d07 d08)))
(assert! '(frame d02 siblings :value (d00 d01 d03 d04 d05 d06 d07 d08)))
(assert! '(frame d03 siblings :value (d00 d01 d02 d04 d05 d06 d07 d08)))
(assert! '(frame d04 siblings :value (d00 d01 d02 d03 d05 d06 d07 d08)))
(assert! '(frame d05 siblings :value (d00 d01 d02 d03 d04 d06 d07 d08)))
(assert! '(frame d06 siblings :value (d00 d01 d02 d03 d04 d05 d07 d08)))
(assert! '(frame d07 siblings :value (d00 d01 d02 d03 d04 d05 d06 d08)))
(assert! '(frame d08 siblings :value (d00 d01 d02 d03 d04 d05 d06 d07)))

(assert! '(frame d14 siblings :value (d15 d16 d17 d18 d19 d20 d21 d22)))
(assert! '(frame d15 siblings :value (d14 d16 d17 d18 d19 d20 d21 d22)))
(assert! '(frame d16 siblings :value (d14 d15 d17 d18 d19 d20 d21 d22)))
(assert! '(frame d17 siblings :value (d14 d15 d16 d18 d19 d20 d21 d22)))
(assert! '(frame d18 siblings :value (d14 d15 d16 d17 d19 d20 d21 d22)))
(assert! '(frame d19 siblings :value (d14 d15 d16 d17 d18 d20 d21 d22)))
(assert! '(frame d20 siblings :value (d14 d15 d16 d17 d18 d19 d21 d22)))
(assert! '(frame d21 siblings :value (d14 d15 d16 d17 d18 d19 d20 d22)))
(assert! '(frame d22 siblings :value (d14 d15 d16 d17 d18 d19 d20 d21)))

(assert! '(frame e00 siblings :value (e01 e02 e03 e04 e05 e06 e07 e08)))
(assert! '(frame e01 siblings :value (e00 e02 e03 e04 e05 e06 e07 e08)))
(assert! '(frame e02 siblings :value (e00 e01 e03 e04 e05 e06 e07 e08)))
(assert! '(frame e03 siblings :value (e00 e01 e02 e04 e05 e06 e07 e08)))
(assert! '(frame e04 siblings :value (e00 e01 e02 e03 e05 e06 e07 e08)))
(assert! '(frame e05 siblings :value (e00 e01 e02 e03 e04 e06 e07 e08)))
(assert! '(frame e06 siblings :value (e00 e01 e02 e03 e04 e05 e07 e08)))
(assert! '(frame e07 siblings :value (e00 e01 e02 e03 e04 e05 e06 e08)))
(assert! '(frame e08 siblings :value (e00 e01 e02 e03 e04 e05 e06 e07)))

(assert! '(frame e14 siblings :value (e15 e16 e17 e18 e19 e20 e21 e22)))
(assert! '(frame e15 siblings :value (e14 e16 e17 e18 e19 e20 e21 e22)))
(assert! '(frame e16 siblings :value (e14 e15 e17 e18 e19 e20 e21 e22)))
(assert! '(frame e17 siblings :value (e14 e15 e16 e18 e19 e20 e21 e22)))
(assert! '(frame e18 siblings :value (e14 e15 e16 e17 e19 e20 e21 e22)))
(assert! '(frame e19 siblings :value (e14 e15 e16 e17 e18 e20 e21 e22)))

```
(assert! '(frame e20 siblings :value (e14 e15 e16 e17 e18 e19 e21 e22)))
(assert! '(frame e21 siblings :value (e14 e15 e16 e17 e18 e19 e20 e22)))
(assert! '(frame e22 siblings :value (e14 e15 e16 e17 e18 e19 e20 e21)))

(assert! '(frame f00 siblings :value (f01 f02 f03 f04 f05 f06 f07 f08)))
(assert! '(frame f01 siblings :value (f00 f02 f03 f04 f05 f06 f07 f08)))
(assert! '(frame f02 siblings :value (f00 f01 f03 f04 f05 f06 f07 f08)))
(assert! '(frame f03 siblings :value (f00 f01 f02 f04 f05 f06 f07 f08)))
(assert! '(frame f04 siblings :value (f00 f01 f02 f03 f05 f06 f07 f08)))
(assert! '(frame f05 siblings :value (f00 f01 f02 f03 f04 f06 f07 f08)))
(assert! '(frame f06 siblings :value (f00 f01 f02 f03 f04 f05 f07 f08)))
(assert! '(frame f07 siblings :value (f00 f01 f02 f03 f04 f05 f06 f08)))
(assert! '(frame f08 siblings :value (f00 f01 f02 f03 f04 f05 f06 f07)))

(assert! '(frame f14 siblings :value (f15 f16 f17 f18 f19 f20 f21 f22)))
(assert! '(frame f15 siblings :value (f14 f16 f17 f18 f19 f20 f21 f22)))
(assert! '(frame f16 siblings :value (f14 f15 f17 f18 f19 f20 f21 f22)))
(assert! '(frame f17 siblings :value (f14 f15 f16 f18 f19 f20 f21 f22)))
(assert! '(frame f18 siblings :value (f14 f15 f16 f17 f19 f20 f21 f22)))
(assert! '(frame f19 siblings :value (f14 f15 f16 f17 f18 f20 f21 f22)))
(assert! '(frame f20 siblings :value (f14 f15 f16 f17 f18 f19 f21 f22)))
(assert! '(frame f21 siblings :value (f14 f15 f16 f17 f18 f19 f20 f22)))
(assert! '(frame f22 siblings :value (f14 f15 f16 f17 f18 f19 f20 f21)))

(assert! '(frame g00 siblings :value (g01 g02 g03 g04 g05 g06 g07 g08)))
(assert! '(frame g01 siblings :value (g00 g02 g03 g04 g05 g06 g07 g08)))
(assert! '(frame g02 siblings :value (g00 g01 g03 g04 g05 g06 g07 g08)))
(assert! '(frame g03 siblings :value (g00 g01 g02 g04 g05 g06 g07 g08)))
(assert! '(frame g04 siblings :value (g00 g01 g02 g03 g05 g06 g07 g08)))
(assert! '(frame g05 siblings :value (g00 g01 g02 g03 g04 g06 g07 g08)))
(assert! '(frame g06 siblings :value (g00 g01 g02 g03 g04 g05 g07 g08)))
(assert! '(frame g07 siblings :value (g00 g01 g02 g03 g04 g05 g06 g08)))
(assert! '(frame g08 siblings :value (g00 g01 g02 g03 g04 g05 g06 g07)))

(assert! '(frame g14 siblings :value (g15 g16 g17 g18 g19 g20 g21 g22)))
(assert! '(frame g15 siblings :value (g14 g16 g17 g18 g19 g20 g21 g22)))
(assert! '(frame g16 siblings :value (g14 g15 g17 g18 g19 g20 g21 g22)))
(assert! '(frame g17 siblings :value (g14 g15 g16 g18 g19 g20 g21 g22)))
(assert! '(frame g18 siblings :value (g14 g15 g16 g17 g19 g20 g21 g22)))
(assert! '(frame g19 siblings :value (g14 g15 g16 g17 g18 g20 g21 g22)))
(assert! '(frame g20 siblings :value (g14 g15 g16 g17 g18 g19 g21 g22)))
(assert! '(frame g21 siblings :value (g14 g15 g16 g17 g18 g19 g20 g22)))
(assert! '(frame g22 siblings :value (g14 g15 g16 g17 g18 g19 g20 g21)))

;;;switch above

(assert! '(frame a02 switch-above :value a01))
(assert! '(frame a03 switch-above :value a01))
```


(assert! '(frame a04 switch-above :value a01))
(assert! '(frame a05 switch-above :value a01))
(assert! '(frame a06 switch-above :value a01))
(assert! '(frame a07 switch-above :value a01))

(assert! '(frame h02 switch-above :value h01))
(assert! '(frame h03 switch-above :value h01))
(assert! '(frame h04 switch-above :value h01))
(assert! '(frame h05 switch-above :value h01))
(assert! '(frame h06 switch-above :value h01))
(assert! '(frame h07 switch-above :value h01))

(assert! '(frame b00 switch-above :value a02))
(assert! '(frame b01 switch-above :value a02))
(assert! '(frame b02 switch-above :value a02))
(assert! '(frame b03 switch-above :value a02))
(assert! '(frame b04 switch-above :value a02))
(assert! '(frame b05 switch-above :value a02))
(assert! '(frame b06 switch-above :value a02))
(assert! '(frame b07 switch-above :value a02))
(assert! '(frame b08 switch-above :value a02))

(assert! '(frame c00 switch-above :value a03))
(assert! '(frame c01 switch-above :value a03))
(assert! '(frame c02 switch-above :value a03))
(assert! '(frame c03 switch-above :value a03))
(assert! '(frame c04 switch-above :value a03))
(assert! '(frame c05 switch-above :value a03))
(assert! '(frame c06 switch-above :value a03))
(assert! '(frame c07 switch-above :value a03))
(assert! '(frame c08 switch-above :value a03))

(assert! '(frame d00 switch-above :value a04))
(assert! '(frame d01 switch-above :value a04))
(assert! '(frame d02 switch-above :value a04))
(assert! '(frame d03 switch-above :value a04))
(assert! '(frame d04 switch-above :value a04))
(assert! '(frame d05 switch-above :value a04))
(assert! '(frame d06 switch-above :value a04))
(assert! '(frame d07 switch-above :value a04))
(assert! '(frame d08 switch-above :value a04))

(assert! '(frame e00 switch-above :value a05))
(assert! '(frame e01 switch-above :value a05))
(assert! '(frame e02 switch-above :value a05))
(assert! '(frame e03 switch-above :value a05))
(assert! '(frame e04 switch-above :value a05))
(assert! '(frame e05 switch-above :value a05))

(assert! '(frame e06 switch-above :value a05))
(assert! '(frame e07 switch-above :value a05))
(assert! '(frame e08 switch-above :value a05))

(assert! '(frame f00 switch-above :value a06))
(assert! '(frame f01 switch-above :value a06))
(assert! '(frame f02 switch-above :value a06))
(assert! '(frame f03 switch-above :value a06))
(assert! '(frame f04 switch-above :value a06))
(assert! '(frame f05 switch-above :value a06))
(assert! '(frame f06 switch-above :value a06))
(assert! '(frame f07 switch-above :value a06))
(assert! '(frame f08 switch-above :value a06))

(assert! '(frame g00 switch-above :value a07))
(assert! '(frame g01 switch-above :value a07))
(assert! '(frame g02 switch-above :value a07))
(assert! '(frame g03 switch-above :value a07))
(assert! '(frame g04 switch-above :value a07))
(assert! '(frame g05 switch-above :value a07))
(assert! '(frame g06 switch-above :value a07))
(assert! '(frame g07 switch-above :value a07))
(assert! '(frame g08 switch-above :value a07))

(assert! '(frame b14 switch-above :value h02))
(assert! '(frame b15 switch-above :value h02))
(assert! '(frame b16 switch-above :value h02))
(assert! '(frame b17 switch-above :value h02))
(assert! '(frame b18 switch-above :value h02))
(assert! '(frame b19 switch-above :value h02))
(assert! '(frame b20 switch-above :value h02))
(assert! '(frame b21 switch-above :value h02))
(assert! '(frame b22 switch-above :value h02))

(assert! '(frame c14 switch-above :value h03))
(assert! '(frame c15 switch-above :value h03))
(assert! '(frame c16 switch-above :value h03))
(assert! '(frame c17 switch-above :value h03))
(assert! '(frame c18 switch-above :value h03))
(assert! '(frame c19 switch-above :value h03))
(assert! '(frame c20 switch-above :value h03))
(assert! '(frame c21 switch-above :value h03))
(assert! '(frame c22 switch-above :value h03))

(assert! '(frame d14 switch-above :value h04))
(assert! '(frame d15 switch-above :value h04))
(assert! '(frame d16 switch-above :value h04))
(assert! '(frame d17 switch-above :value h04))

```
(assert! '(frame d18 switch-above :value h04))
(assert! '(frame d19 switch-above :value h04))
(assert! '(frame d20 switch-above :value h04))
(assert! '(frame d21 switch-above :value h04))
(assert! '(frame d22 switch-above :value h04))

(assert! '(frame e14 switch-above :value h05))
(assert! '(frame e15 switch-above :value h05))
(assert! '(frame e16 switch-above :value h05))
(assert! '(frame e17 switch-above :value h05))
(assert! '(frame e18 switch-above :value h05))
(assert! '(frame e19 switch-above :value h05))
(assert! '(frame e20 switch-above :value h05))
(assert! '(frame e21 switch-above :value h05))
(assert! '(frame e22 switch-above :value h05))

(assert! '(frame f14 switch-above :value h06))
(assert! '(frame f15 switch-above :value h06))
(assert! '(frame f16 switch-above :value h06))
(assert! '(frame f17 switch-above :value h06))
(assert! '(frame f18 switch-above :value h06))
(assert! '(frame f19 switch-above :value h06))
(assert! '(frame f20 switch-above :value h06))
(assert! '(frame f21 switch-above :value h06))
(assert! '(frame f22 switch-above :value h06))

(assert! '(frame g14 switch-above :value h07))
(assert! '(frame g15 switch-above :value h07))
(assert! '(frame g16 switch-above :value h07))
(assert! '(frame g17 switch-above :value h07))
(assert! '(frame g18 switch-above :value h07))
(assert! '(frame g19 switch-above :value h07))
(assert! '(frame g20 switch-above :value h07))
(assert! '(frame g21 switch-above :value h07))
(assert! '(frame g22 switch-above :value h07))

;;; switches-below

(assert! '(frame a01 switches-below :value (a02 a03 a04 a05 a06 a07)))

(assert! '(frame h01 switches-below :value (h02 h03 h04 h05 h06 h07)))

(assert! '(frame a02 switches-below :value
  (b00 b01 b02 b03 b04 b05 b06 b07 b08)))
(assert! '(frame a03 switches-below :value
  (c00 c01 c02 c03 c04 c05 c06 c07 c08)))
(assert! '(frame a04 switches-below :value
  (d00 d01 d02 d03 d04 d05 d06 d07 d08)))
```

```
(assert! '(frame a05 switches-below :value
  (e00 e01 e02 e03 e04 e05 e06 e07 e08)))
(assert! '(frame a06 switches-below :value
  (f00 f01 f02 f03 f04 f05 f06 f07 f08)))
(assert! '(frame a07 switches-below :value
  (g00 g01 g02 g03 g04 g05 g06 g07 g08)))
(assert! '(frame h02 switches-below :value
  (b14 b15 b16 b17 b18 b19 b20 b21 b22)))
(assert! '(frame h03 switches-below :value
  (c14 c15 c16 c17 c18 c19 c20 c21 c22)))
(assert! '(frame h04 switches-below :value
  (d14 d15 d16 d17 d18 d19 d20 d21 d22)))
(assert! '(frame h05 switches-below :value
  (e14 e15 e16 e17 e18 e19 e20 e21 e22)))
(assert! '(frame h06 switches-below :value
  (f14 f15 f16 f17 f18 f19 f20 f21 f22)))
(assert! '(frame h07 switches-below :value
  (g14 g15 g16 g17 g18 g19 g20 g21 g22)))
```

;;; corresponding-sensors

```
(assert! '(frame a01 corresponding-sensors :value (am0 am1)))
(assert! '(frame a02 corresponding-sensors :value (am2 bm0)))
(assert! '(frame a03 corresponding-sensors :value (am3 cm0)))
(assert! '(frame a04 corresponding-sensors :value (am4 dm0)))
(assert! '(frame a05 corresponding-sensors :value (am5 em0)))
(assert! '(frame a06 corresponding-sensors :value (am6 fm0)))
(assert! '(frame a07 corresponding-sensors :value (am7 gm0)))
```

```
(assert! '(frame h01 corresponding-sensors :value (hm0 hm1)))
(assert! '(frame h02 corresponding-sensors :value (hm2 bm1)))
(assert! '(frame h03 corresponding-sensors :value (hm3 cm1)))
(assert! '(frame h04 corresponding-sensors :value (hm4 dm1)))
(assert! '(frame h05 corresponding-sensors :value (hm5 em1)))
(assert! '(frame h06 corresponding-sensors :value (hm6 fm1)))
(assert! '(frame h07 corresponding-sensors :value (hm7 gm1)))
```

;;; sensor-above and top-sensor

```
(assert! '(frame a01 sensor-above am0))
(assert! '(frame a01 top-sensor am0))
(dolist (switch '(a02 a03 a04 a05 a06 a07))
  (assert! '(frame ,switch sensor-above :value am1))
  (assert! '(frame ,switch top-sensor :value am0)))
```

```
(assert! '(frame h01 sensor-above hm0))
(assert! '(frame h01 top-sensor hm0))
(dolist (switch '(h02 h03 h04 h05 h06 h07))
```

```
(assert! '(frame ,switch sensor-above :value hm1))
(assert! '(frame ,switch top-sensor :value hm0)))

(dolist (switch '(b00 b01 b02 b03 b04 b05 b06 b07 b08))
  (assert! '(frame ,switch sensor-above :value bm0))
  (assert! '(frame ,switch top-sensor :value am0)))

(dolist (switch '(b14 b15 b16 b17 b18 b19 b20 b21 b22))
  (assert! '(frame ,switch sensor-above :value bm1))
  (assert! '(frame ,switch top-sensor :value hm0)))

(dolist (switch '(c00 c01 c02 c03 c04 c05 c06 c07 c08))
  (assert! '(frame ,switch sensor-above :value cm0))
  (assert! '(frame ,switch top-sensor :value am0)))

(dolist (switch '(c14 c15 c16 c17 c18 c19 c20 c21 c22))
  (assert! '(frame ,switch sensor-above :value cm1))
  (assert! '(frame ,switch top-sensor :value hm0)))

(dolist (switch '(d00 d01 d02 d03 d04 d05 d06 d07 d08))
  (assert! '(frame ,switch sensor-above :value dm0))
  (assert! '(frame ,switch top-sensor :value am0)))

(dolist (switch '(d14 d15 d16 d17 d18 d19 d20 d21 d22))
  (assert! '(frame ,switch sensor-above :value dm1))
  (assert! '(frame ,switch top-sensor :value hm0)))

(dolist (switch '(e00 e01 e02 e03 e04 e05 e06 e07 e08))
  (assert! '(frame ,switch sensor-above :value em0))
  (assert! '(frame ,switch top-sensor :value am0)))

(dolist (switch '(e14 e15 e16 e17 e18 e19 e20 e21 e22))
  (assert! '(frame ,switch sensor-above :value em1))
  (assert! '(frame ,switch top-sensor :value hm0)))

(dolist (switch '(f00 f01 f02 f03 f04 f05 f06 f07 f08))
  (assert! '(frame ,switch sensor-above :value fm0))
  (assert! '(frame ,switch top-sensor :value am0)))

(dolist (switch '(f14 f15 f16 f17 f18 f19 f20 f21 f22))
  (assert! '(frame ,switch sensor-above :value fm1))
  (assert! '(frame ,switch top-sensor :value hm0)))

(dolist (switch '(g00 g01 g02 g03 g04 g05 g06 g07 g08))
  (assert! '(frame ,switch sensor-above :value gm0))
  (assert! '(frame ,switch top-sensor :value am0)))

(dolist (switch '(g14 g15 g16 g17 g18 g19 g20 g21 g22))
```

```
(assert! '(frame ,switch sensor-above :value gmi))
(assert! '(frame ,switch top-sensor :value hm0)))

;;; redundant switches

(assert! '(frame b00 redundant-switch :value b14))
(assert! '(frame b01 redundant-switch :value b15))
(assert! '(frame b02 redundant-switch :value b16))
(assert! '(frame b03 redundant-switch :value b17))
(assert! '(frame b04 redundant-switch :value b18))
(assert! '(frame b05 redundant-switch :value b19))
(assert! '(frame b06 redundant-switch :value b20))
(assert! '(frame b07 redundant-switch :value b21))
(assert! '(frame b08 redundant-switch :value b22))

(assert! '(frame b14 redundant-switch :value b00))
(assert! '(frame b15 redundant-switch :value b01))
(assert! '(frame b16 redundant-switch :value b02))
(assert! '(frame b17 redundant-switch :value b03))
(assert! '(frame b18 redundant-switch :value b04))
(assert! '(frame b19 redundant-switch :value b05))
(assert! '(frame b20 redundant-switch :value b06))
(assert! '(frame b21 redundant-switch :value b07))
(assert! '(frame b22 redundant-switch :value b08))

(assert! '(frame c00 redundant-switch :value c14))
(assert! '(frame c01 redundant-switch :value c15))
(assert! '(frame c02 redundant-switch :value c16))
(assert! '(frame c03 redundant-switch :value c17))
(assert! '(frame c04 redundant-switch :value c18))
(assert! '(frame c05 redundant-switch :value c19))
(assert! '(frame c06 redundant-switch :value c20))
(assert! '(frame c07 redundant-switch :value c21))
(assert! '(frame c08 redundant-switch :value c22))

(assert! '(frame c14 redundant-switch :value c00))
(assert! '(frame c15 redundant-switch :value c01))
(assert! '(frame c16 redundant-switch :value c02))
(assert! '(frame c17 redundant-switch :value c03))
(assert! '(frame c18 redundant-switch :value c04))
(assert! '(frame c19 redundant-switch :value c05))
(assert! '(frame c20 redundant-switch :value c06))
(assert! '(frame c21 redundant-switch :value c07))
(assert! '(frame c22 redundant-switch :value c08))

(assert! '(frame d00 redundant-switch :value d14))
(assert! '(frame d01 redundant-switch :value d15))
(assert! '(frame d02 redundant-switch :value d16))
```

```
(assert! '(frame d03 redundant-switch :value d17))
(assert! '(frame d04 redundant-switch :value d18))
(assert! '(frame d05 redundant-switch :value d19))
(assert! '(frame d06 redundant-switch :value d20))
(assert! '(frame d07 redundant-switch :value d21))
(assert! '(frame d08 redundant-switch :value d22))

(assert! '(frame d14 redundant-switch :value d00))
(assert! '(frame d15 redundant-switch :value d01))
(assert! '(frame d16 redundant-switch :value d02))
(assert! '(frame d17 redundant-switch :value d03))
(assert! '(frame d18 redundant-switch :value d04))
(assert! '(frame d19 redundant-switch :value d05))
(assert! '(frame d20 redundant-switch :value d06))
(assert! '(frame d21 redundant-switch :value d07))
(assert! '(frame d22 redundant-switch :value d08))

(assert! '(frame e00 redundant-switch :value e14))
(assert! '(frame e01 redundant-switch :value e15))
(assert! '(frame e02 redundant-switch :value e16))
(assert! '(frame e03 redundant-switch :value e17))
(assert! '(frame e04 redundant-switch :value e18))
(assert! '(frame e05 redundant-switch :value e19))
(assert! '(frame e06 redundant-switch :value e20))
(assert! '(frame e07 redundant-switch :value e21))
(assert! '(frame e08 redundant-switch :value e22))

(assert! '(frame e14 redundant-switch :value e00))
(assert! '(frame e15 redundant-switch :value e01))
(assert! '(frame e16 redundant-switch :value e02))
(assert! '(frame e17 redundant-switch :value e03))
(assert! '(frame e18 redundant-switch :value e04))
(assert! '(frame e19 redundant-switch :value e05))
(assert! '(frame e20 redundant-switch :value e06))
(assert! '(frame e21 redundant-switch :value e07))
(assert! '(frame e22 redundant-switch :value e08))

(assert! '(frame f00 redundant-switch :value f14))
(assert! '(frame f01 redundant-switch :value f15))
(assert! '(frame f02 redundant-switch :value f16))
(assert! '(frame f03 redundant-switch :value f17))
(assert! '(frame f04 redundant-switch :value f18))
(assert! '(frame f05 redundant-switch :value f19))
(assert! '(frame f06 redundant-switch :value f20))
(assert! '(frame f07 redundant-switch :value f21))
(assert! '(frame f08 redundant-switch :value f22))

(assert! '(frame f14 redundant-switch :value f00))
```

```
(assert! '(frame f15 redundant-switch :value f01))
(assert! '(frame f16 redundant-switch :value f02))
(assert! '(frame f17 redundant-switch :value f03))
(assert! '(frame f18 redundant-switch :value f04))
(assert! '(frame f19 redundant-switch :value f05))
(assert! '(frame f20 redundant-switch :value f06))
(assert! '(frame f21 redundant-switch :value f07))
(assert! '(frame f22 redundant-switch :value f08))
```

```
(assert! '(frame g00 redundant-switch :value g14))
(assert! '(frame g01 redundant-switch :value g15))
(assert! '(frame g02 redundant-switch :value g16))
(assert! '(frame g03 redundant-switch :value g17))
(assert! '(frame g04 redundant-switch :value g18))
(assert! '(frame g05 redundant-switch :value g19))
(assert! '(frame g06 redundant-switch :value g20))
(assert! '(frame g07 redundant-switch :value g21))
(assert! '(frame g08 redundant-switch :value g22))
```

```
(assert! '(frame g14 redundant-switch :value g00))
(assert! '(frame g15 redundant-switch :value g01))
(assert! '(frame g16 redundant-switch :value g02))
(assert! '(frame g17 redundant-switch :value g03))
(assert! '(frame g18 redundant-switch :value g04))
(assert! '(frame g19 redundant-switch :value g05))
(assert! '(frame g20 redundant-switch :value g06))
(assert! '(frame g21 redundant-switch :value g07))
(assert! '(frame g22 redundant-switch :value g08))
```

;;; nodes for the soft fault network now.

```
(fcreate-instance 'node 'a-node1)
(fcreate-instance 'node 'a-node2)
(fcreate-instance 'node 'a-node3)
(fcreate-instance 'node 'a-node4)
(fcreate-instance 'node 'a-node5)
(fcreate-instance 'node 'a-node6)
(fcreate-instance 'node 'a-node7)
(fcreate-instance 'node 'a-node8)
```

```
(fcreate-instance 'node 'h-node1)
(fcreate-instance 'node 'h-node2)
(fcreate-instance 'node 'h-node3)
(fcreate-instance 'node 'h-node4)
(fcreate-instance 'node 'h-node5)
(fcreate-instance 'node 'h-node6)
(fcreate-instance 'node 'h-node7)
(fcreate-instance 'node 'h-node8)
```



```
(fcreate-instance 'node 'b-node1)
(fcreate-instance 'node 'b-node2)

(fcreate-instance 'node 'c-node1)
(fcreate-instance 'node 'c-node2)

(fcreate-instance 'node 'd-node1)
(fcreate-instance 'node 'd-node2)

(fcreate-instance 'node 'e-node1)
(fcreate-instance 'node 'e-node2)

(fcreate-instance 'node 'f-node1)
(fcreate-instance 'node 'f-node2)

(fcreate-instance 'node 'g-node1)
(fcreate-instance 'node 'g-node2)

;;; fill in the nodes.

(assert! '(frame a-node1 lower-switches :value (a01)))
(assert! '(frame a-node1 upper-sensor :value am0))
(assert! '(frame a-node1 lower-sensors :value (am1)))
(assert! '(frame a-node1 lower-nodes :value (a-node2)))

(assert! '(frame h-node1 lower-switches :value (h01)))
(assert! '(frame h-node1 upper-sensor :value hm0))
(assert! '(frame h-node1 lower-sensors :value (hm1)))
(assert! '(frame h-node1 lower-nodes :value (h-node2)))

(assert! '(frame a-node2 upper-switch :value a01))
(assert! '(frame a-node2 lower-switches :value (a02 a03 a04 a05 a06 a07)))
(assert! '(frame a-node2 upper-sensor :value am1))
(assert! '(frame a-node2 lower-sensors :value (am2 am3 am4 am5 am6 am7)))
(assert! '(frame a-node2 upper-node :value a-node1))
(assert! '(frame a-node2 lower-nodes :value
  (a-node3 a-node4 a-node5 a-node6 a-node7 a-node8)))

(assert! '(frame h-node2 upper-switch :value h01))
(assert! '(frame h-node2 lower-switches :value (h02 h03 h04 h05 h06 h07)))
(assert! '(frame h-node2 upper-sensor :value hm1))
(assert! '(frame h-node2 lower-sensors :value (hm2 hm3 hm4 hm5 hm6 hm7)))
(assert! '(frame h-node2 upper-node :value h-node1))
(assert! '(frame h-node2 lower-nodes :value
  (h-node3 h-node4 h-node5 h-node6 h-node7 h-node8)))

(assert! '(frame a-node3 upper-switch :value a02))
```

```
(assert! '(frame a-node3 lower-switches :value
  (b00 b01 b02 b03 b04 b05 b06 b07 b08)))
(assert! '(frame a-node3 upper-sensor :value am2))
(assert! '(frame a-node3 lower-sensors :value (bm0)))
(assert! '(frame a-node3 upper-node :value a-node2))
(assert! '(frame a-node3 lower-nodes :value (b-node1)))

(assert! '(frame a-node4 upper-switch :value a03))
(assert! '(frame a-node4 lower-switches :value
  (c00 c01 c02 c03 c04 c05 c06 c07 c08)))
(assert! '(frame a-node4 upper-sensor :value am3))
(assert! '(frame a-node4 lower-sensors :value (cm0)))
(assert! '(frame a-node4 upper-node :value a-node2))
(assert! '(frame a-node4 lower-nodes :value (c-node1)))

(assert! '(frame a-node5 upper-switch :value a04))
(assert! '(frame a-node5 lower-switches :value
  (d00 d01 d02 d03 d04 d05 d06 d07 d08)))
(assert! '(frame a-node5 upper-sensor :value am4))
(assert! '(frame a-node5 lower-sensors :value (dm0)))
(assert! '(frame a-node5 upper-node :value a-node2))
(assert! '(frame a-node5 lower-nodes :value (d-node1)))

(assert! '(frame a-node6 upper-switch :value a05))
(assert! '(frame a-node6 lower-switches :value
  (e00 e01 e02 e03 e04 e05 e06 e07 e08)))
(assert! '(frame a-node6 upper-sensor :value am5))
(assert! '(frame a-node6 lower-sensors :value (em0)))
(assert! '(frame a-node6 upper-node :value a-node2))
(assert! '(frame a-node6 lower-nodes :value (e-node1)))

(assert! '(frame a-node7 upper-switch :value a06))
(assert! '(frame a-node7 lower-switches :value
  (f00 f01 f02 f03 f04 f05 f06 f07 f08)))
(assert! '(frame a-node7 upper-sensor :value am6))
(assert! '(frame a-node7 lower-sensors :value (fm0)))
(assert! '(frame a-node7 upper-node :value a-node2))
(assert! '(frame a-node7 lower-nodes :value (f-node1)))

(assert! '(frame a-node8 upper-switch :value a07))
(assert! '(frame a-node8 lower-switches :value
  (g00 g01 g02 g03 g04 g05 g06 g07 g08)))
(assert! '(frame a-node8 upper-sensor :value am7))
(assert! '(frame a-node8 lower-sensors :value (gm0)))
(assert! '(frame a-node8 upper-node :value a-node2))
(assert! '(frame a-node8 lower-nodes :value (g-node1)))

(assert! '(frame h-node3 upper-switch :value h02))
```

```
(assert! '(frame h-node3 lower-switches :value
  (b14 b15 b16 b17 b18 b19 b20 b21 b22)))
(assert! '(frame h-node3 upper-sensor :value hm2))
(assert! '(frame h-node3 lower-sensors :value (bm1)))
(assert! '(frame h-node3 upper-node :value h-node2))
(assert! '(frame h-node3 lower-nodes :value (b-node2)))

(assert! '(frame h-node4 upper-switch :value h03))
(assert! '(frame h-node4 lower-switches :value
  (c14 c15 c16 c17 c18 c19 c20 c21 c22)))
(assert! '(frame h-node4 upper-sensor :value hm3))
(assert! '(frame h-node4 lower-sensors :value (cm1)))
(assert! '(frame h-node4 upper-node :value h-node2))
(assert! '(frame h-node4 lower-nodes :value (c-node2)))

(assert! '(frame h-node5 upper-switch :value h04))
(assert! '(frame h-node5 lower-switches :value
  (d14 d15 d16 d17 d18 d19 d20 d21 d22)))
(assert! '(frame h-node5 upper-sensor :value hm4))
(assert! '(frame h-node5 lower-sensors :value (dm1)))
(assert! '(frame h-node5 upper-node :value h-node2))
(assert! '(frame h-node5 lower-nodes :value (d-node2)))

(assert! '(frame h-node6 upper-switch :value h05))
(assert! '(frame h-node6 lower-switches :value
  (e14 e15 e16 e17 e18 e19 e20 e21 e22)))
(assert! '(frame h-node6 upper-sensor :value hm5))
(assert! '(frame h-node6 lower-sensors :value (em1)))
(assert! '(frame h-node6 upper-node :value h-node2))
(assert! '(frame h-node6 lower-nodes :value (e-node2)))

(assert! '(frame h-node7 upper-switch :value h06))
(assert! '(frame h-node7 lower-switches :value
  (f14 f15 f16 f17 f18 f19 f20 f21 f22)))
(assert! '(frame h-node7 upper-sensor :value hm6))
(assert! '(frame h-node7 lower-sensors :value (fm1)))
(assert! '(frame h-node7 upper-node :value h-node2))
(assert! '(frame h-node7 lower-nodes :value (f-node2)))

(assert! '(frame h-node8 upper-switch :value h07))
(assert! '(frame h-node8 lower-switches :value
  (g14 g15 g16 g17 g18 g19 g20 g21 g22)))
(assert! '(frame h-node8 upper-sensor :value hm7))
(assert! '(frame h-node8 lower-sensors :value (gm1)))
(assert! '(frame h-node8 upper-node :value h-node2))
(assert! '(frame h-node8 lower-nodes :value (g-node2)))

(assert! '(frame b-node1 upper-switch :value a02))
```

```
(assert! '(frame b-node1 lower-switches :value
  (b00 b01 b02 b03 b04 b05 b06 b07 b08)))
(assert! '(frame b-node1 upper-sensor :value bm0))
(assert! '(frame b-node1 upper-node :value a-node3))

(assert! '(frame b-node2 upper-switch :value h02))
(assert! '(frame b-node2 lower-switches :value
  (b14 b15 b16 b17 b18 b19 b20 b21 b22)))
(assert! '(frame b-node2 upper-sensor :value bm1))
(assert! '(frame b-node2 upper-node :value h-node3))

(assert! '(frame c-node1 upper-switch :value a03))
(assert! '(frame c-node1 lower-switches :value
  (c00 c01cb02 c03 c04 c05 c06 c07 c08)))
(assert! '(frame c-node1 upper-sensor :value cm0))
(assert! '(frame c-node1 upper-node :value a-node4))

(assert! '(frame c-node2 upper-switch :value h03))
(assert! '(frame c-node2 lower-switches :value
  (c14 c15 c16 c17 c18 c19 c20 c21 c22)))
(assert! '(frame c-node2 upper-sensor :value cm1))
(assert! '(frame c-node2 upper-node :value h-node4))

(assert! '(frame d-node1 upper-switch :value a04))
(assert! '(frame d-node1 lower-switches :value
  (d00 d01db02 d03 d04 d05 d06 d07 d08)))
(assert! '(frame d-node1 upper-sensor :value dm0))
(assert! '(frame d-node1 upper-node :value a-node5))

(assert! '(frame d-node2 upper-switch :value h04))
(assert! '(frame d-node2 lower-switches :value
  (d14 d15 d16 d17 d18 d19 d20 d21 d22)))
(assert! '(frame d-node2 upper-sensor :value dm1))
(assert! '(frame d-node2 upper-node :value h-node5))

(assert! '(frame e-node1 upper-switch :value a05))
(assert! '(frame e-node1 lower-switches :value
  (e00 e01eb02 e03 e04 e05 e06 e07 e08)))
(assert! '(frame e-node1 upper-sensor :value em0))
(assert! '(frame e-node1 upper-node :value a-node6))

(assert! '(frame e-node2 upper-switch :value h05))
(assert! '(frame e-node2 lower-switches :value
  (e14 e15 e16 e17 e18 e19 e20 e21 e22)))
(assert! '(frame e-node2 upper-sensor :value em1))
(assert! '(frame e-node2 upper-node :value h-node6))

(assert! '(frame f-node1 upper-switch :value a06))
```

```
(assert! '(frame f-node1 lower-switches :value
(f00 f01fb02 f03 f04 f05 f06 f07 f08)))
(assert! '(frame f-node1 upper-sensor :value fm0))
(assert! '(frame f-node1 upper-node :value a-node7))

(assert! '(frame f-node2 upper-switch :value h06))
(assert! '(frame f-node2 lower-switches :value
(f14 f15 f16 f17 f18 f19 f20 f21 f22)))
(assert! '(frame f-node2 upper-sensor :value fm1))
(assert! '(frame f-node2 upper-node :value h-node7))

(assert! '(frame g-node1 upper-switch :value a07))
(assert! '(frame g-node1 lower-switches :value
(g00 g01gb02 g03 g04 g05 g06 g07 g08)))
(assert! '(frame g-node1 upper-sensor :value gm0))
(assert! '(frame g-node1 upper-node :value a-node8))

(assert! '(frame g-node2 upper-switch :value h07))
(assert! '(frame g-node2 lower-switches :value
(g14 g15 g16 g17 g18 g19 g20 g21 g22)))
(assert! '(frame g-node2 upper-sensor :value gm1))
(assert! '(frame g-node2 upper-node :value h-node8))

(dolist (switch '(b00 b01 b02 b03 b04 b05 b06 b07 b08))
(assert! '(frame ,switch upper-node :value a-node3)))

(dolist (switch '(b14 b15 b16 b17 b18 b19 b20 b21 b22))
(assert! '(frame ,switch upper-node :value h-node3)))

(dolist (switch '(c00 c01 c02 c03 c04 c05 c06 c07 c08))
(assert! '(frame ,switch upper-node :value a-node4)))

(dolist (switch '(c14 c15 c16 c17 c18 c19 c20 c21 c22))
(assert! '(frame ,switch upper-node :value h-node4)))

(dolist (switch '(d00 d01 d02 d03 d04 d05 d06 d07 d08))
(assert! '(frame ,switch upper-node :value a-node5)))

(dolist (switch '(d14 d15 d16 d17 d18 d19 d20 d21 d22))
(assert! '(frame ,switch upper-node :value h-node5)))

(dolist (switch '(e00 e01 e02 e03 e04 e05 e06 e07 e08))
(assert! '(frame ,switch upper-node :value a-node6)))

(dolist (switch '(e14 e15 e16 e17 e18 e19 e20 e21 e22))
(assert! '(frame ,switch upper-node :value h-node6)))

(dolist (switch '(f00 f01 f02 f03 f04 f05 f06 f07 f08))
```

```
(assert! '(frame ,switch upper-node :value a-node7)))

(dolist (switch '(f14 f15 f16 f17 f18 f19 f20 f21 f22))
  (assert! '(frame ,switch upper-node :value h-node7)))

(dolist (switch '(g00 g01 g02 g03 g04 g05 g06 g07 g08))
  (assert! '(frame ,switch upper-node :value a-node8)))

(dolist (switch '(g14 g15 g16 g17 g18 g19 g20 g21 g22))
  (assert! '(frame ,switch upper-node :value h-node8)))

(assert! '(frame am0 lower-node :value a-node1))
(assert! '(frame am1 upper-node :value a-node1))
(assert! '(frame am1 lower-node :value a-node2))

(assert! '(frame hm0 lower-node :value h-node1))
(assert! '(frame hm1 upper-node :value h-node1))
(assert! '(frame hm1 lower-node :value h-node2))

(dolist (sensor '(am2 am3 am4 am5 am6 am7))
  (assert! '(frame ,sensor upper-node :value a-node2)))

(dolist (sensor '(hm2 hm3 hm4 hm5 hm6 hm7))
  (assert! '(frame ,sensor upper-node :value h-node2)))

(assert! '(frame am2 lower-node :value a-node3))
(assert! '(frame am3 lower-node :value a-node4))
(assert! '(frame am4 lower-node :value a-node5))
(assert! '(frame am5 lower-node :value a-node6))
(assert! '(frame am6 lower-node :value a-node7))
(assert! '(frame am7 lower-node :value a-node8))

(assert! '(frame hm2 lower-node :value h-node3))
(assert! '(frame hm3 lower-node :value h-node4))
(assert! '(frame hm4 lower-node :value h-node5))
(assert! '(frame hm5 lower-node :value h-node6))
(assert! '(frame hm6 lower-node :value h-node7))
(assert! '(frame hm7 lower-node :value h-node8))

(assert! '(frame bm0 upper-node :value a-node3))
(assert! '(frame bm0 lower-node :value b-node1))
(assert! '(frame bm1 upper-node :value h-node3))
(assert! '(frame bm1 lower-node :value b-node2))

(assert! '(frame cm0 upper-node :value a-node4))
(assert! '(frame cm0 lower-node :value c-node1))
(assert! '(frame cm1 upper-node :value h-node4))
```

```
(assert! '(frame cm1 lower-node :value c-node2))

(assert! '(frame dm0 upper-node :value a-node5))
(assert! '(frame dm0 lower-node :value d-node1))
(assert! '(frame dm1 upper-node :value h-node5))
(assert! '(frame dm1 lower-node :value d-node2))

(assert! '(frame em0 upper-node :value a-node6))
(assert! '(frame em0 lower-node :value e-node1))
(assert! '(frame em1 upper-node :value h-node6))
(assert! '(frame em1 lower-node :value e-node2))

(assert! '(frame fm0 upper-node :value a-node7))
(assert! '(frame fm0 lower-node :value f-node1))
(assert! '(frame fm1 upper-node :value h-node7))
(assert! '(frame fm1 lower-node :value f-node2))

(assert! '(frame gm0 upper-node :value a-node8))
(assert! '(frame gm0 lower-node :value g-node1))
(assert! '(frame gm1 upper-node :value h-node8))
(assert! '(frame gm1 lower-node :value g-node2))

;;; LLPs

(dolist (llp '(llp-a llp-b llp-c llp-d llp-e llp-f llp-g llp-h))
  (fcreate-instance 'llp llp)
  (assert! '(frame ,llp name :value ,llp)))

(assert! '(frame llp-a contained-switches-bus-a :value
  (a01 a02 a03 a04 a05 a06 a07)))
(assert! '(frame llp-a contained-switches-bus-b :value ()))
(assert! '(frame llp-a contained-sensors :value
  (am0 am1 am2 am3 am4 am5 am6 am7)))

(assert! '(frame llp-h contained-switches-bus-a :value ()))
(assert! '(frame llp-h contained-switches-bus-b :value
  (h01 h02 h03 h04 h05 h06 h07)))
(assert! '(frame llp-h contained-sensors :value
  (hm0 hm1 hm2 hm3 hm4 hm5 hm6 hm7)))

(assert! '(frame llp-b contained-switches-bus-a :value
  (b00 b01 b02 b03 b04 b05 b06 b07 b08)))
(assert! '(frame llp-b contained-switches-bus-b :value
  (b14 b15 b16 b17 b18 b19 b20 b21 b22)))
(assert! '(frame llp-b contained-sensors :value (bm0 bm1)))

(assert! '(frame llp-c contained-switches-bus-a :value
  (c00 c01 c02 c03 c04 c05 c06 c07 c08)))
```

```
(assert! '(frame llp-c contained-switches-bus-b :value
(c14 c15 c16 c17 c18 c19 c20 c21 c22)))
(assert! '(frame llp-c contained-sensors :value (cm0 cm1)))

(assert! '(frame llp-d contained-switches-bus-a :value
(d00 d01 d02 d03 d04 d05 d06 d07 d08)))
(assert! '(frame llp-d contained-switches-bus-b :value
(d14 d15 d16 d17 d18 d19 d20 d21 d22)))
(assert! '(frame llp-d contained-sensors :value (dm0 dm1)))

(assert! '(frame llp-e contained-switches-bus-a :value
(e00 e01 e02 e03 e04 e05 e06 e07 e08)))
(assert! '(frame llp-e contained-switches-bus-b :value
(e14 e15 e16 e17 e18 e19 e20 e21 e22)))
(assert! '(frame llp-e contained-sensors :value (em0 em1)))

(assert! '(frame llp-f contained-switches-bus-a :value
(f00 f01 f02 f03 f04 f05 f06 f07 f08)))
(assert! '(frame llp-f contained-switches-bus-b :value
(f14 f15 f16 f17 f18 f19 f20 f21 f22)))
(assert! '(frame llp-f contained-sensors :value (fm0 fm1)))

(assert! '(frame llp-g contained-switches-bus-a :value
(g00 g01 g02 g03 g04 g05 g06 g07 g08)))
(assert! '(frame llp-g contained-switches-bus-b :value
(g14 g15 g16 g17 g18 g19 g20 g21 g22)))
(assert! '(frame llp-g contained-sensors :value (gm0 gm1)))

;;; cables

(fcreate-instance 'cable 'llp-a-cable-0)
(assert! '(frame llp-a-cable-0 name :value llp-a-cable-0))
(assert! '(frame llp-a-cable-0 in :value source-a))
(assert! '(frame llp-a-cable-0 out :value (a01)))

(fcreate-instance 'cable 'llp-a-cable-1)
(assert! '(frame llp-a-cable-1 name :value llp-a-cable-1))
(assert! '(frame llp-a-cable-1 in :value a01))
(assert! '(frame llp-a-cable-1 out :value (a02 a03 a04 a05 a06 a07)))

(fcreate-instance 'cable 'llp-h-cable-0)
(assert! '(frame llp-h-cable-0 name :value llp-h-cable-0))
(assert! '(frame llp-h-cable-0 in :value source-b))
(assert! '(frame llp-h-cable-0 out :value (h01)))

(fcreate-instance 'cable 'llp-h-cable-1)
(assert! '(frame llp-h-cable-1 name :value llp-h-cable-1))
(assert! '(frame llp-h-cable-1 in :value h01))
```



```
(assert! '(frame llp-h-cable-1 out :value (h02 h03 h04 h05 h06 h07)))

(fcreate-instance 'cable 'llp-a-cable-2)
(assert! '(frame llp-a-cable-2 name :value llp-a-cable-2))
(assert! '(frame llp-a-cable-2 in :value a02))
(assert! '(frame llp-a-cable-2 out :value
  (b00 b01 b02 b03 b04 b05 b06 b07 b08)))

(fcreate-instance 'cable 'llp-a-cable-3)
(assert! '(frame llp-a-cable-3 name :value llp-a-cable-3))
(assert! '(frame llp-a-cable-3 in :value a03))
(assert! '(frame llp-a-cable-3 out :value
  (c00 c01 c02 c03 c04 c05 c06 c07 c08)))

(fcreate-instance 'cable 'llp-a-cable-4)
(assert! '(frame llp-a-cable-4 name :value llp-a-cable-4))
(assert! '(frame llp-a-cable-4 in :value a04))
(assert! '(frame llp-a-cable-4 out :value
  (d00 d01 d02 d03 d04 d05 d06 d07 d08)))

(fcreate-instance 'cable 'llp-a-cable-5)
(assert! '(frame llp-a-cable-5 name :value llp-a-cable-5))
(assert! '(frame llp-a-cable-5 in :value a05))
(assert! '(frame llp-a-cable-5 out :value
  (e00 e01 e02 e03 e04 e05 e06 e07 e08)))

(fcreate-instance 'cable 'llp-a-cable-6)
(assert! '(frame llp-a-cable-6 name :value llp-a-cable-6))
(assert! '(frame llp-a-cable-6 in :value a06))
(assert! '(frame llp-a-cable-6 out :value
  (f00 f01 f02 f03 f04 f05 f06 f07 f08)))

(fcreate-instance 'cable 'llp-a-cable-7)
(assert! '(frame llp-a-cable-7 name :value llp-a-cable-7))
(assert! '(frame llp-a-cable-7 in :value a07))
(assert! '(frame llp-a-cable-7 out :value
  (g00 g01 g02 g03 g04 g05 g06 g07 g08)))

(fcreate-instance 'cable 'llp-h-cable-2)
(assert! '(frame llp-h-cable-2 name :value llp-h-cable-2))
(assert! '(frame llp-h-cable-2 in :value h02))
(assert! '(frame llp-h-cable-2 out :value
  (b14 b15 b16 b17 b18 b19 b20 b21 b22)))

(fcreate-instance 'cable 'llp-h-cable-3)
(assert! '(frame llp-h-cable-3 name :value llp-h-cable-3))
(assert! '(frame llp-h-cable-3 in :value h03))
(assert! '(frame llp-h-cable-3 out :value
```

```
(c14 c15 c16 c17 c18 c19 c20 c21 c22)))

(fcreate-instance 'cable 'llp-h-cable-4)
(assert! '(frame llp-h-cable-4 name :value llp-h-cable-4))
(assert! '(frame llp-h-cable-4 in :value h04))
(assert! '(frame llp-h-cable-4 out :value
  (d14 d15 d16 d17 d18 d19 d20 d21 d22)))

(fcreate-instance 'cable 'llp-h-cable-5)
(assert! '(frame llp-h-cable-5 name :value llp-h-cable-5))
(assert! '(frame llp-h-cable-5 in :value h05))
(assert! '(frame llp-h-cable-5 out :value
  (e14 e15 e16 e17 e18 e19 e20 e21 e22)))

(fcreate-instance 'cable 'llp-h-cable-6)
(assert! '(frame llp-h-cable-6 name :value llp-h-cable-6))
(assert! '(frame llp-h-cable-6 in :value h06))
(assert! '(frame llp-h-cable-6 out :value
  (f14 f15 f16 f17 f18 f19 f20 f21 f22)))

(fcreate-instance 'cable 'llp-h-cable-7)
(assert! '(frame llp-h-cable-7 name :value llp-h-cable-7))
(assert! '(frame llp-h-cable-7 in :value h07))
(assert! '(frame llp-h-cable-7 out :value
  (g14 g15 g16 g17 g18 g19 g20 g21 g22)))

(fcreate-instance 'cable 'llp-b-cable-0)
(assert! '(frame llp-b-cable-0 name :value llp-b-cable-0))
(assert! '(frame llp-b-cable-0 in :value b00))
(assert! '(frame llp-b-cable-0 out :value (load-b00)))

(fcreate-instance 'cable 'llp-b-cable-1)
(assert! '(frame llp-b-cable-1 name :value llp-b-cable-1))
(assert! '(frame llp-b-cable-1 in :value b01))
(assert! '(frame llp-b-cable-1 out :value (load-b01)))

(fcreate-instance 'cable 'llp-b-cable-2)
(assert! '(frame llp-b-cable-2 name :value llp-b-cable-2))
(assert! '(frame llp-b-cable-2 in :value b02))
(assert! '(frame llp-b-cable-2 out :value (load-b02)))

(fcreate-instance 'cable 'llp-b-cable-3)
(assert! '(frame llp-b-cable-3 name :value llp-b-cable-3))
(assert! '(frame llp-b-cable-3 in :value b03))
(assert! '(frame llp-b-cable-3 out :value (load-b03)))

(fcreate-instance 'cable 'llp-b-cable-4)
(assert! '(frame llp-b-cable-4 name :value llp-b-cable-4))
```

```
(assert! '(frame llp-b-cable-4 in :value b04))
(assert! '(frame llp-b-cable-4 out :value (load-b04)))

(fcreate-instance 'cable 'llp-b-cable-5)
(assert! '(frame llp-b-cable-5 name :value llp-b-cable-5))
(assert! '(frame llp-b-cable-5 in :value b05))
(assert! '(frame llp-b-cable-5 out :value (load-b05)))

(fcreate-instance 'cable 'llp-b-cable-6)
(assert! '(frame llp-b-cable-6 name :value llp-b-cable-6))
(assert! '(frame llp-b-cable-6 in :value b06))
(assert! '(frame llp-b-cable-6 out :value (load-b06)))

(fcreate-instance 'cable 'llp-b-cable-7)
(assert! '(frame llp-b-cable-7 name :value llp-b-cable-7))
(assert! '(frame llp-b-cable-7 in :value b07))
(assert! '(frame llp-b-cable-7 out :value (load-b07)))

(fcreate-instance 'cable 'llp-b-cable-8)
(assert! '(frame llp-b-cable-8 name :value llp-b-cable-8))
(assert! '(frame llp-b-cable-8 in :value b08))
(assert! '(frame llp-b-cable-8 out :value (load-b08)))

(fcreate-instance 'cable 'llp-b-cable-14)
(assert! '(frame llp-b-cable-14 name :value llp-b-cable-14))
(assert! '(frame llp-b-cable-14 in :value b14))
(assert! '(frame llp-b-cable-14 out :value (load-b14)))

(fcreate-instance 'cable 'llp-b-cable-15)
(assert! '(frame llp-b-cable-15 name :value llp-b-cable-15))
(assert! '(frame llp-b-cable-15 in :value b15))
(assert! '(frame llp-b-cable-15 out :value (load-b15)))

(fcreate-instance 'cable 'llp-b-cable-16)
(assert! '(frame llp-b-cable-16 name :value llp-b-cable-16))
(assert! '(frame llp-b-cable-16 in :value b16))
(assert! '(frame llp-b-cable-16 out :value (load-b16)))

(fcreate-instance 'cable 'llp-b-cable-17)
(assert! '(frame llp-b-cable-17 name :value llp-b-cable-17))
(assert! '(frame llp-b-cable-17 in :value b17))
(assert! '(frame llp-b-cable-17 out :value (load-b17)))

(fcreate-instance 'cable 'llp-b-cable-18)
(assert! '(frame llp-b-cable-18 name :value llp-b-cable-18))
(assert! '(frame llp-b-cable-18 in :value b18))
(assert! '(frame llp-b-cable-18 out :value (load-b18)))
```

```
(fcreate-instance 'cable 'llp-b-cable-19)
(assert! '(frame llp-b-cable-19 name :value llp-b-cable-19))
(assert! '(frame llp-b-cable-19 in :value b19))
(assert! '(frame llp-b-cable-19 out :value (load-b19)))

(fcreate-instance 'cable 'llp-b-cable-20)
(assert! '(frame llp-b-cable-20 name :value llp-b-cable-20))
(assert! '(frame llp-b-cable-20 in :value b20))
(assert! '(frame llp-b-cable-20 out :value (load-b20)))

(fcreate-instance 'cable 'llp-b-cable-21)
(assert! '(frame llp-b-cable-21 name :value llp-b-cable-21))
(assert! '(frame llp-b-cable-21 in :value b21))
(assert! '(frame llp-b-cable-21 out :value (load-b21)))

(fcreate-instance 'cable 'llp-b-cable-22)
(assert! '(frame llp-b-cable-22 name :value llp-b-cable-22))
(assert! '(frame llp-b-cable-22 in :value b22))
(assert! '(frame llp-b-cable-22 out :value (load-b22)))

(fcreate-instance 'cable 'llp-c-cable-0)
(assert! '(frame llp-c-cable-0 name :value llp-c-cable-0))
(assert! '(frame llp-c-cable-0 in :value c00))
(assert! '(frame llp-c-cable-0 out :value (load-c00)))

(fcreate-instance 'cable 'llp-c-cable-1)
(assert! '(frame llp-c-cable-1 name :value llp-c-cable-1))
(assert! '(frame llp-c-cable-1 in :value c01))
(assert! '(frame llp-c-cable-1 out :value (load-c01)))

(fcreate-instance 'cable 'llp-c-cable-2)
(assert! '(frame llp-c-cable-2 name :value llp-c-cable-2))
(assert! '(frame llp-c-cable-2 in :value c02))
(assert! '(frame llp-c-cable-2 out :value (load-c02)))

(fcreate-instance 'cable 'llp-c-cable-3)
(assert! '(frame llp-c-cable-3 name :value llp-c-cable-3))
(assert! '(frame llp-c-cable-3 in :value c03))
(assert! '(frame llp-c-cable-3 out :value (load-c03)))

(fcreate-instance 'cable 'llp-c-cable-4)
(assert! '(frame llp-c-cable-4 name :value llp-c-cable-4))
(assert! '(frame llp-c-cable-4 in :value c04))
(assert! '(frame llp-c-cable-4 out :value (load-c04)))

(fcreate-instance 'cable 'llp-c-cable-5)
(assert! '(frame llp-c-cable-5 name :value llp-c-cable-5))
(assert! '(frame llp-c-cable-5 in :value c05))
```

```
(assert! '(frame llp-c-cable-5 out :value (load-c05)))

(fcreate-instance 'cable 'llp-c-cable-6)
(assert! '(frame llp-c-cable-6 name :value llp-c-cable-6))
(assert! '(frame llp-c-cable-6 in :value c06))
(assert! '(frame llp-c-cable-6 out :value (load-c06)))

(fcreate-instance 'cable 'llp-c-cable-7)
(assert! '(frame llp-c-cable-7 name :value llp-c-cable-7))
(assert! '(frame llp-c-cable-7 in :value c07))
(assert! '(frame llp-c-cable-7 out :value (load-c07)))

(fcreate-instance 'cable 'llp-c-cable-8)
(assert! '(frame llp-c-cable-8 name :value llp-c-cable-8))
(assert! '(frame llp-c-cable-8 in :value c08))
(assert! '(frame llp-c-cable-8 out :value (load-c08)))

(fcreate-instance 'cable 'llp-c-cable-14)
(assert! '(frame llp-c-cable-14 name :value llp-c-cable-14))
(assert! '(frame llp-c-cable-14 in :value c14))
(assert! '(frame llp-c-cable-14 out :value (load-c14)))

(fcreate-instance 'cable 'llp-c-cable-15)
(assert! '(frame llp-c-cable-15 name :value llp-c-cable-15))
(assert! '(frame llp-c-cable-15 in :value c15))
(assert! '(frame llp-c-cable-15 out :value (load-c15)))

(fcreate-instance 'cable 'llp-c-cable-16)
(assert! '(frame llp-c-cable-16 name :value llp-c-cable-16))
(assert! '(frame llp-c-cable-16 in :value c16))
(assert! '(frame llp-c-cable-16 out :value (load-c16)))

(fcreate-instance 'cable 'llp-c-cable-17)
(assert! '(frame llp-c-cable-17 name :value llp-c-cable-17))
(assert! '(frame llp-c-cable-17 in :value c17))
(assert! '(frame llp-c-cable-17 out :value (load-c17)))

(fcreate-instance 'cable 'llp-c-cable-18)
(assert! '(frame llp-c-cable-18 name :value llp-c-cable-18))
(assert! '(frame llp-c-cable-18 in :value c18))
(assert! '(frame llp-c-cable-18 out :value (load-c18)))

(fcreate-instance 'cable 'llp-c-cable-19)
(assert! '(frame llp-c-cable-19 name :value llp-c-cable-19))
(assert! '(frame llp-c-cable-19 in :value c19))
(assert! '(frame llp-c-cable-19 out :value (load-c19)))

(fcreate-instance 'cable 'llp-c-cable-20)
```

```
(assert! '(frame llp-c-cable-20 name :value llp-c-cable-20))
(assert! '(frame llp-c-cable-20 in :value c20))
(assert! '(frame llp-c-cable-20 out :value (load-c20)))

(fcreate-instance 'cable 'llp-c-cable-21)
(assert! '(frame llp-c-cable-21 name :value llp-c-cable-21))
(assert! '(frame llp-c-cable-21 in :value c21))
(assert! '(frame llp-c-cable-21 out :value (load-c21)))

(fcreate-instance 'cable 'llp-c-cable-22)
(assert! '(frame llp-c-cable-22 name :value llp-c-cable-22))
(assert! '(frame llp-c-cable-22 in :value c22))
(assert! '(frame llp-c-cable-22 out :value (load-c22)))

(fcreate-instance 'cable 'llp-d-cable-0)
(assert! '(frame llp-d-cable-0 name :value llp-d-cable-0))
(assert! '(frame llp-d-cable-0 in :value d00))
(assert! '(frame llp-d-cable-0 out :value (load-d00)))

(fcreate-instance 'cable 'llp-d-cable-1)
(assert! '(frame llp-d-cable-1 name :value llp-d-cable-1))
(assert! '(frame llp-d-cable-1 in :value d01))
(assert! '(frame llp-d-cable-1 out :value (load-d01)))

(fcreate-instance 'cable 'llp-d-cable-2)
(assert! '(frame llp-d-cable-2 name :value llp-d-cable-2))
(assert! '(frame llp-d-cable-2 in :value d02))
(assert! '(frame llp-d-cable-2 out :value (load-d02)))

(fcreate-instance 'cable 'llp-d-cable-3)
(assert! '(frame llp-d-cable-3 name :value llp-d-cable-3))
(assert! '(frame llp-d-cable-3 in :value d03))
(assert! '(frame llp-d-cable-3 out :value (load-d03)))

(fcreate-instance 'cable 'llp-d-cable-4)
(assert! '(frame llp-d-cable-4 name :value llp-d-cable-4))
(assert! '(frame llp-d-cable-4 in :value d04))
(assert! '(frame llp-d-cable-4 out :value (load-d04)))

(fcreate-instance 'cable 'llp-d-cable-5)
(assert! '(frame llp-d-cable-5 name :value llp-d-cable-5))
(assert! '(frame llp-d-cable-5 in :value d05))
(assert! '(frame llp-d-cable-5 out :value (load-d05)))

(fcreate-instance 'cable 'llp-d-cable-6)
(assert! '(frame llp-d-cable-6 name :value llp-d-cable-6))
(assert! '(frame llp-d-cable-6 in :value d06))
(assert! '(frame llp-d-cable-6 out :value (load-d06)))
```

```
(fcreate-instance 'cable 'llp-d-cable-7)
(assert! '(frame llp-d-cable-7 name :value llp-d-cable-7))
(assert! '(frame llp-d-cable-7 in :value d07))
(assert! '(frame llp-d-cable-7 out :value (load-d07)))

(fcreate-instance 'cable 'llp-d-cable-8)
(assert! '(frame llp-d-cable-8 name :value llp-d-cable-8))
(assert! '(frame llp-d-cable-8 in :value d08))
(assert! '(frame llp-d-cable-8 out :value (load-d08)))

(fcreate-instance 'cable 'llp-d-cable-14)
(assert! '(frame llp-d-cable-14 name :value llp-d-cable-14))
(assert! '(frame llp-d-cable-14 in :value d14))
(assert! '(frame llp-d-cable-14 out :value (load-d14)))

(fcreate-instance 'cable 'llp-d-cable-15)
(assert! '(frame llp-d-cable-15 name :value llp-d-cable-15))
(assert! '(frame llp-d-cable-15 in :value d15))
(assert! '(frame llp-d-cable-15 out :value (load-d15)))

(fcreate-instance 'cable 'llp-d-cable-16)
(assert! '(frame llp-d-cable-16 name :value llp-d-cable-16))
(assert! '(frame llp-d-cable-16 in :value d16))
(assert! '(frame llp-d-cable-16 out :value (load-d16)))

(fcreate-instance 'cable 'llp-d-cable-17)
(assert! '(frame llp-d-cable-17 name :value llp-d-cable-17))
(assert! '(frame llp-d-cable-17 in :value d17))
(assert! '(frame llp-d-cable-17 out :value (load-d17)))

(fcreate-instance 'cable 'llp-d-cable-18)
(assert! '(frame llp-d-cable-18 name :value llp-d-cable-18))
(assert! '(frame llp-d-cable-18 in :value d18))
(assert! '(frame llp-d-cable-18 out :value (load-d18)))

(fcreate-instance 'cable 'llp-d-cable-19)
(assert! '(frame llp-d-cable-19 name :value llp-d-cable-19))
(assert! '(frame llp-d-cable-19 in :value d19))
(assert! '(frame llp-d-cable-19 out :value (load-d19)))

(fcreate-instance 'cable 'llp-d-cable-20)
(assert! '(frame llp-d-cable-20 name :value llp-d-cable-20))
(assert! '(frame llp-d-cable-20 in :value d20))
(assert! '(frame llp-d-cable-20 out :value (load-d20)))

(fcreate-instance 'cable 'llp-d-cable-21)
(assert! '(frame llp-d-cable-21 name :value llp-d-cable-21))
```

```
(assert! '(frame llp-d-cable-21 in :value d21))
(assert! '(frame llp-d-cable-21 out :value (load-d21)))

(fcreate-instance 'cable 'llp-d-cable-22)
(assert! '(frame llp-d-cable-22 name :value llp-d-cable-22))
(assert! '(frame llp-d-cable-22 in :value d22))
(assert! '(frame llp-d-cable-22 out :value (load-d22)))

(fcreate-instance 'cable 'llp-e-cable-0)
(assert! '(frame llp-e-cable-0 name :value llp-e-cable-0))
(assert! '(frame llp-e-cable-0 in :value e00))
(assert! '(frame llp-e-cable-0 out :value (load-e00)))

(fcreate-instance 'cable 'llp-e-cable-1)
(assert! '(frame llp-e-cable-1 name :value llp-e-cable-1))
(assert! '(frame llp-e-cable-1 in :value e01))
(assert! '(frame llp-e-cable-1 out :value (load-e01)))

(fcreate-instance 'cable 'llp-e-cable-2)
(assert! '(frame llp-e-cable-2 name :value llp-e-cable-2))
(assert! '(frame llp-e-cable-2 in :value e02))
(assert! '(frame llp-e-cable-2 out :value (load-e02)))

(fcreate-instance 'cable 'llp-e-cable-3)
(assert! '(frame llp-e-cable-3 name :value llp-e-cable-3))
(assert! '(frame llp-e-cable-3 in :value e03))
(assert! '(frame llp-e-cable-3 out :value (load-e03)))

(fcreate-instance 'cable 'llp-e-cable-4)
(assert! '(frame llp-e-cable-4 name :value llp-e-cable-4))
(assert! '(frame llp-e-cable-4 in :value e04))
(assert! '(frame llp-e-cable-4 out :value (load-e04)))

(fcreate-instance 'cable 'llp-e-cable-5)
(assert! '(frame llp-e-cable-5 name :value llp-e-cable-5))
(assert! '(frame llp-e-cable-5 in :value e05))
(assert! '(frame llp-e-cable-5 out :value (load-e05)))

(fcreate-instance 'cable 'llp-e-cable-6)
(assert! '(frame llp-e-cable-6 name :value llp-e-cable-6))
(assert! '(frame llp-e-cable-6 in :value e06))
(assert! '(frame llp-e-cable-6 out :value (load-e06)))

(fcreate-instance 'cable 'llp-e-cable-7)
(assert! '(frame llp-e-cable-7 name :value llp-e-cable-7))
(assert! '(frame llp-e-cable-7 in :value e07))
(assert! '(frame llp-e-cable-7 out :value (load-e07)))
```



```
(fcreate-instance 'cable 'llp-e-cable-8)
(assert! '(frame llp-e-cable-8 name :value llp-e-cable-8))
(assert! '(frame llp-e-cable-8 in :value e08))
(assert! '(frame llp-e-cable-8 out :value (load-e08)))

(fcreate-instance 'cable 'llp-e-cable-14)
(assert! '(frame llp-e-cable-14 name :value llp-e-cable-14))
(assert! '(frame llp-e-cable-14 in :value e14))
(assert! '(frame llp-e-cable-14 out :value (load-e14)))

(fcreate-instance 'cable 'llp-e-cable-15)
(assert! '(frame llp-e-cable-15 name :value llp-e-cable-15))
(assert! '(frame llp-e-cable-15 in :value e15))
(assert! '(frame llp-e-cable-15 out :value (load-e15)))

(fcreate-instance 'cable 'llp-e-cable-16)
(assert! '(frame llp-e-cable-16 name :value llp-e-cable-16))
(assert! '(frame llp-e-cable-16 in :value e16))
(assert! '(frame llp-e-cable-16 out :value (load-e16)))

(fcreate-instance 'cable 'llp-e-cable-17)
(assert! '(frame llp-e-cable-17 name :value llp-e-cable-17))
(assert! '(frame llp-e-cable-17 in :value e17))
(assert! '(frame llp-e-cable-17 out :value (load-e17)))

(fcreate-instance 'cable 'llp-e-cable-18)
(assert! '(frame llp-e-cable-18 name :value llp-e-cable-18))
(assert! '(frame llp-e-cable-18 in :value e18))
(assert! '(frame llp-e-cable-18 out :value (load-e18)))

(fcreate-instance 'cable 'llp-e-cable-19)
(assert! '(frame llp-e-cable-19 name :value llp-e-cable-19))
(assert! '(frame llp-e-cable-19 in :value e19))
(assert! '(frame llp-e-cable-19 out :value (load-e19)))

(fcreate-instance 'cable 'llp-e-cable-20)
(assert! '(frame llp-e-cable-20 name :value llp-e-cable-20))
(assert! '(frame llp-e-cable-20 in :value e20))
(assert! '(frame llp-e-cable-20 out :value (load-e20)))

(fcreate-instance 'cable 'llp-e-cable-21)
(assert! '(frame llp-e-cable-21 name :value llp-e-cable-21))
(assert! '(frame llp-e-cable-21 in :value e21))
(assert! '(frame llp-e-cable-21 out :value (load-e21)))

(fcreate-instance 'cable 'llp-e-cable-22)
(assert! '(frame llp-e-cable-22 name :value llp-e-cable-22))
(assert! '(frame llp-e-cable-22 in :value e22))
```

```
(assert! '(frame llp-e-cable-22 out :value (load-e22)))

(fcreate-instance 'cable 'llp-f-cable-0)
(assert! '(frame llp-f-cable-0 name :value llp-f-cable-0))
(assert! '(frame llp-f-cable-0 in :value f00))
(assert! '(frame llp-f-cable-0 out :value (load-f00)))

(fcreate-instance 'cable 'llp-f-cable-1)
(assert! '(frame llp-f-cable-1 name :value llp-f-cable-1))
(assert! '(frame llp-f-cable-1 in :value f01))
(assert! '(frame llp-f-cable-1 out :value (load-f01)))

(fcreate-instance 'cable 'llp-f-cable-2)
(assert! '(frame llp-f-cable-2 name :value llp-f-cable-2))
(assert! '(frame llp-f-cable-2 in :value f02))
(assert! '(frame llp-f-cable-2 out :value (load-f02)))

(fcreate-instance 'cable 'llp-f-cable-3)
(assert! '(frame llp-f-cable-3 name :value llp-f-cable-3))
(assert! '(frame llp-f-cable-3 in :value f03))
(assert! '(frame llp-f-cable-3 out :value (load-f03)))

(fcreate-instance 'cable 'llp-f-cable-4)
(assert! '(frame llp-f-cable-4 name :value llp-f-cable-4))
(assert! '(frame llp-f-cable-4 in :value f04))
(assert! '(frame llp-f-cable-4 out :value (load-f04)))

(fcreate-instance 'cable 'llp-f-cable-5)
(assert! '(frame llp-f-cable-5 name :value llp-f-cable-5))
(assert! '(frame llp-f-cable-5 in :value f05))
(assert! '(frame llp-f-cable-5 out :value (load-f05)))

(fcreate-instance 'cable 'llp-f-cable-6)
(assert! '(frame llp-f-cable-6 name :value llp-f-cable-6))
(assert! '(frame llp-f-cable-6 in :value f06))
(assert! '(frame llp-f-cable-6 out :value (load-f06)))

(fcreate-instance 'cable 'llp-f-cable-7)
(assert! '(frame llp-f-cable-7 name :value llp-f-cable-7))
(assert! '(frame llp-f-cable-7 in :value f07))
(assert! '(frame llp-f-cable-7 out :value (load-f07)))

(fcreate-instance 'cable 'llp-f-cable-8)
(assert! '(frame llp-f-cable-8 name :value llp-f-cable-8))
(assert! '(frame llp-f-cable-8 in :value f08))
(assert! '(frame llp-f-cable-8 out :value (load-f08)))

(fcreate-instance 'cable 'llp-f-cable-14)
```

```
(assert! '(frame llp-f-cable-14 name :value llp-f-cable-14))
(assert! '(frame llp-f-cable-14 in :value f14))
(assert! '(frame llp-f-cable-14 out :value (load-f14)))

(fcreate-instance 'cable 'llp-f-cable-15)
(assert! '(frame llp-f-cable-15 name :value llp-f-cable-15))
(assert! '(frame llp-f-cable-15 in :value f15))
(assert! '(frame llp-f-cable-15 out :value (load-f15)))

(fcreate-instance 'cable 'llp-f-cable-16)
(assert! '(frame llp-f-cable-16 name :value llp-f-cable-16))
(assert! '(frame llp-f-cable-16 in :value f16))
(assert! '(frame llp-f-cable-16 out :value (load-f16)))

(fcreate-instance 'cable 'llp-f-cable-17)
(assert! '(frame llp-f-cable-17 name :value llp-f-cable-17))
(assert! '(frame llp-f-cable-17 in :value f17))
(assert! '(frame llp-f-cable-17 out :value (load-f17)))

(fcreate-instance 'cable 'llp-f-cable-18)
(assert! '(frame llp-f-cable-18 name :value llp-f-cable-18))
(assert! '(frame llp-f-cable-18 in :value f18))
(assert! '(frame llp-f-cable-18 out :value (load-f18)))

(fcreate-instance 'cable 'llp-f-cable-19)
(assert! '(frame llp-f-cable-19 name :value llp-f-cable-19))
(assert! '(frame llp-f-cable-19 in :value f19))
(assert! '(frame llp-f-cable-19 out :value (load-f19)))

(fcreate-instance 'cable 'llp-f-cable-20)
(assert! '(frame llp-f-cable-20 name :value llp-f-cable-20))
(assert! '(frame llp-f-cable-20 in :value f20))
(assert! '(frame llp-f-cable-20 out :value (load-f20)))

(fcreate-instance 'cable 'llp-f-cable-21)
(assert! '(frame llp-f-cable-21 name :value llp-f-cable-21))
(assert! '(frame llp-f-cable-21 in :value f21))
(assert! '(frame llp-f-cable-21 out :value (load-f21)))

(fcreate-instance 'cable 'llp-f-cable-22)
(assert! '(frame llp-f-cable-22 name :value llp-f-cable-22))
(assert! '(frame llp-f-cable-22 in :value f22))
(assert! '(frame llp-f-cable-22 out :value (load-f22)))

(fcreate-instance 'cable 'llp-g-cable-0)
(assert! '(frame llp-g-cable-0 name :value llp-g-cable-0))
(assert! '(frame llp-g-cable-0 in :value g00))
(assert! '(frame llp-g-cable-0 out :value (load-g00)))
```

```
(fcreate-instance 'cable 'llp-g-cable-1)
(assert! '(frame llp-g-cable-1 name :value llp-g-cable-1))
(assert! '(frame llp-g-cable-1 in :value g01))
(assert! '(frame llp-g-cable-1 out :value (load-g01)))

(fcreate-instance 'cable 'llp-g-cable-2)
(assert! '(frame llp-g-cable-2 name :value llp-g-cable-2))
(assert! '(frame llp-g-cable-2 in :value g02))
(assert! '(frame llp-g-cable-2 out :value (load-g02)))

(fcreate-instance 'cable 'llp-g-cable-3)
(assert! '(frame llp-g-cable-3 name :value llp-g-cable-3))
(assert! '(frame llp-g-cable-3 in :value g03))
(assert! '(frame llp-g-cable-3 out :value (load-g03)))

(fcreate-instance 'cable 'llp-g-cable-4)
(assert! '(frame llp-g-cable-4 name :value llp-g-cable-4))
(assert! '(frame llp-g-cable-4 in :value g04))
(assert! '(frame llp-g-cable-4 out :value (load-g04)))

(fcreate-instance 'cable 'llp-g-cable-5)
(assert! '(frame llp-g-cable-5 name :value llp-g-cable-5))
(assert! '(frame llp-g-cable-5 in :value g05))
(assert! '(frame llp-g-cable-5 out :value (load-g05)))

(fcreate-instance 'cable 'llp-g-cable-6)
(assert! '(frame llp-g-cable-6 name :value llp-g-cable-6))
(assert! '(frame llp-g-cable-6 in :value g06))
(assert! '(frame llp-g-cable-6 out :value (load-g06)))

(fcreate-instance 'cable 'llp-g-cable-7)
(assert! '(frame llp-g-cable-7 name :value llp-g-cable-7))
(assert! '(frame llp-g-cable-7 in :value g07))
(assert! '(frame llp-g-cable-7 out :value (load-g07)))

(fcreate-instance 'cable 'llp-g-cable-8)
(assert! '(frame llp-g-cable-8 name :value llp-g-cable-8))
(assert! '(frame llp-g-cable-8 in :value g08))
(assert! '(frame llp-g-cable-8 out :value (load-g08)))

(fcreate-instance 'cable 'llp-g-cable-14)
(assert! '(frame llp-g-cable-14 name :value llp-g-cable-14))
(assert! '(frame llp-g-cable-14 in :value g14))
(assert! '(frame llp-g-cable-14 out :value (load-g14)))

(fcreate-instance 'cable 'llp-g-cable-15)
(assert! '(frame llp-g-cable-15 name :value llp-g-cable-15))
```

```
(assert! '(frame llp-g-cable-15 in :value g15))
(assert! '(frame llp-g-cable-15 out :value (load-g15)))

(fcreate-instance 'cable 'llp-g-cable-16)
(assert! '(frame llp-g-cable-16 name :value llp-g-cable-16))
(assert! '(frame llp-g-cable-16 in :value g16))
(assert! '(frame llp-g-cable-16 out :value (load-g16)))

(fcreate-instance 'cable 'llp-g-cable-17)
(assert! '(frame llp-g-cable-17 name :value llp-g-cable-17))
(assert! '(frame llp-g-cable-17 in :value g17))
(assert! '(frame llp-g-cable-17 out :value (load-g17)))

(fcreate-instance 'cable 'llp-g-cable-18)
(assert! '(frame llp-g-cable-18 name :value llp-g-cable-18))
(assert! '(frame llp-g-cable-18 in :value g18))
(assert! '(frame llp-g-cable-18 out :value (load-g18)))

(fcreate-instance 'cable 'llp-g-cable-19)
(assert! '(frame llp-g-cable-19 name :value llp-g-cable-19))
(assert! '(frame llp-g-cable-19 in :value g19))
(assert! '(frame llp-g-cable-19 out :value (load-g19)))

(fcreate-instance 'cable 'llp-g-cable-20)
(assert! '(frame llp-g-cable-20 name :value llp-g-cable-20))
(assert! '(frame llp-g-cable-20 in :value g20))
(assert! '(frame llp-g-cable-20 out :value (load-g20)))

(fcreate-instance 'cable 'llp-g-cable-21)
(assert! '(frame llp-g-cable-21 name :value llp-g-cable-21))
(assert! '(frame llp-g-cable-21 in :value g21))
(assert! '(frame llp-g-cable-21 out :value (load-g21)))

(fcreate-instance 'cable 'llp-g-cable-22)
(assert! '(frame llp-g-cable-22 name :value llp-g-cable-22))
(assert! '(frame llp-g-cable-22 in :value g22))
(assert! '(frame llp-g-cable-22 out :value (load-g22)))

;;; loads

(dolist (load '((load-b00 llp-b-cable-0) (load-b01 llp-b-cable-1)
               (load-b02 llp-b-cable-2) (load-b03 llp-b-cable-3)
               (load-b04 llp-b-cable-4) (load-b05 llp-b-cable-5)
               (load-b06 llp-b-cable-6) (load-b07 llp-b-cable-7)
               (load-b08 llp-b-cable-8) (load-b14 llp-b-cable-14)
               (load-b15 llp-b-cable-15) (load-b16 llp-b-cable-16)
               (load-b17 llp-b-cable-17) (load-b18 llp-b-cable-18)
               (load-b19 llp-b-cable-19) (load-b20 llp-b-cable-20))
```

```
(load-b21 llp-b-cable-21) (load-b22 llp-b-cable-22)))
  (fcreate-instance 'load (car load))
  (assert! '(frame ,(car load) llp :value llp-b))
  (assert! '(frame ,(car load) cable-in :value ,(cadr load))))

(dolist (load '((load-c00 llp-c-cable-0) (load-c01 llp-c-cable-1)
  (load-c02 llp-c-cable-2) (load-c03 llp-c-cable-3)
  (load-c04 llp-c-cable-4) (load-c05 llp-c-cable-5)
  (load-c06 llp-c-cable-6) (load-c07 llp-c-cable-7)
  (load-c08 llp-c-cable-8) (load-c14 llp-c-cable-14)
  (load-c15 llp-c-cable-15) (load-c16 llp-c-cable-16)
  (load-c17 llp-c-cable-17) (load-c18 llp-c-cable-18)
  (load-c19 llp-c-cable-19) (load-c20 llp-c-cable-20)
  (load-c21 llp-c-cable-21) (load-c22 llp-c-cable-22)))
  (fcreate-instance 'load (car load))
  (assert! '(frame ,(car load) llp :value llp-c))
  (assert! '(frame ,(car load) cable-in :value ,(cadr load))))

(dolist (load '((load-d00 llp-d-cable-0) (load-d01 llp-d-cable-1)
  (load-d02 llp-d-cable-2) (load-d03 llp-d-cable-3)
  (load-d04 llp-d-cable-4) (load-d05 llp-d-cable-5)
  (load-d06 llp-d-cable-6) (load-d07 llp-d-cable-7)
  (load-d08 llp-d-cable-8) (load-d14 llp-d-cable-14)
  (load-d15 llp-d-cable-15) (load-d16 llp-d-cable-16)
  (load-d17 llp-d-cable-17) (load-d18 llp-d-cable-18)
  (load-d19 llp-d-cable-19) (load-d20 llp-d-cable-20)
  (load-d21 llp-d-cable-21) (load-d22 llp-d-cable-22)))
  (fcreate-instance 'load (car load))
  (assert! '(frame ,(car load) llp :value llp-d))
  (assert! '(frame ,(car load) cable-in :value ,(cadr load))))

(dolist (load '((load-e00 llp-e-cable-0) (load-e01 llp-e-cable-1)
  (load-e02 llp-e-cable-2) (load-e03 llp-e-cable-3)
  (load-e04 llp-e-cable-4) (load-e05 llp-e-cable-5)
  (load-e06 llp-e-cable-6) (load-e07 llp-e-cable-7)
  (load-e08 llp-e-cable-8) (load-e14 llp-e-cable-14)
  (load-e15 llp-e-cable-15) (load-e16 llp-e-cable-16)
  (load-e17 llp-e-cable-17) (load-e18 llp-e-cable-18)
  (load-e19 llp-e-cable-19) (load-e20 llp-e-cable-20)
  (load-e21 llp-e-cable-21) (load-e22 llp-e-cable-22)))
  (fcreate-instance 'load (car load))
  (assert! '(frame ,(car load) llp :value llp-e))
  (assert! '(frame ,(car load) cable-in :value ,(cadr load))))

(dolist (load '((load-f00 llp-f-cable-0) (load-f01 llp-f-cable-1)
  (load-f02 llp-f-cable-2) (load-f03 llp-f-cable-3)
  (load-f04 llp-f-cable-4) (load-f05 llp-f-cable-5)
  (load-f06 llp-f-cable-6) (load-f07 llp-f-cable-7)
```

```
(load-f08 llp-f-cable-8)(load-f14 llp-f-cable-14)
(load-f15 llp-f-cable-15) (load-f16 llp-f-cable-16)
(load-f17 llp-f-cable-17) (load-f18 llp-f-cable-18)
(load-f19 llp-f-cable-19) (load-f20 llp-f-cable-20)
(load-f21 llp-f-cable-21) (load-f22 llp-f-cable-22)))
(fcreate-instance 'load (car load))
(assert! '(frame ,(car load) llp :value llp-f))
(assert! '(frame ,(car load) cable-in :value ,(cadr load))))

(dolist (load '((load-g00 llp-g-cable-0) (load-g01 llp-g-cable-1)
(load-g02 llp-g-cable-2) (load-g03 llp-g-cable-3)
(load-g04 llp-g-cable-4) (load-g05 llp-g-cable-5)
(load-g06 llp-g-cable-6) (load-g07 llp-g-cable-7)
(load-g08 llp-g-cable-8) (load-g14 llp-g-cable-14)
(load-g15 llp-g-cable-15) (load-g16 llp-g-cable-16)
(load-g17 llp-g-cable-17) (load-g18 llp-g-cable-18)
(load-g19 llp-g-cable-19) (load-g20 llp-g-cable-20)
(load-g21 llp-g-cable-21) (load-g22 llp-g-cable-22))))
(fcreate-instance 'load (car load))
(assert! '(frame ,(car load) llp :value llp-g))
(assert! '(frame ,(car load) cable-in :value ,(cadr load))))

;;; sources

(fcreate-instance 'source 'source-a)
(assert! '(frame source-a name :value source-a))
(assert! '(frame source-a voltage :value 120))
(assert! '(frame source-a cable-out :value llp-a-cable-1))

(fcreate-instance 'source 'source-b)
(assert! '(frame source-b name :value source-b))
(assert! '(frame source-b voltage :value 120))
(assert! '(frame source-b cable-out :value llp-h-cable-1))

;;; bring up the user interface

(user::run-frames)
```

4.3.2 The Hard Fault Expert System

The hard fault expert system for FRAMES is given here. This expert system is organized as three rule groups as follows:

Multiple Fault Control Rule Group

cc

cc The Control Rule Group

```
00
00 This rule group controls the analysis of symptom sets by
00 calling the hard fault and diagnosis rule groups on symptom sets.
00
00 There is a problem in this rule group.
00 The rules need to run for every symptom set of the symptom set queue.
00 However, the semantics of THERE EXISTS is to only fire once, as is
00 the semantics of simple rules. Until new and better semantics for
00 the needed kind of control is defined, we kludge this by resetting
00 the fired? slot on rules every time cluster-symptoms is called.
00
```

RULE-GROUP : mf-control

```
CONTROL : ((start (mf-control-rule1))
(mf-control-rule1 (mf-control-rule2))
(mf-control-rule2 (mf-control-rule3 mf-control-rule4))
(mf-control-rule3 (mf-control-rule1))
(mf-control-rule4 (mf-control-rule1)))
```

```
00
00 This rule watches for symptom sets in the symptom set queue.
00 This allows new symptom sets to arrive during diagnosis of
00 existing symptoms sets.
00
```

```
MF-Control-Rule1
THERE EXISTS symptom-set in symptom-set-queue
< ::>
[ the-symptom-set = symptom-set ] >
```

```
;
00
00 When we have a symptom set we first cluster the symptom sets.
00 A symptom set may be the result of multiple independent faults,
00 hopefully from different parts of the power system topology.
00 Clustering the symptom set will produce one symptom
00 set from each identifiably distinct location in the power system
00 that is bus-wise independent (trying to recognize separate faults).
00
```

```
00 This rule calls the hard fault rule group to determine a diagnosis
00 for each cluster (a symptom set) in the-symptom-set.
00 This consists of setting the
00 symptom-set, setting multiple-hard-fault-analysis flag and
00 initializing diagnosis-set. The multiple-hard-fault rule group
00 is called followed by the mf-diagnosis rule group.
00
```

```
MF-Control-Rule2
FOR ALL cluster in power-domain :: cluster-symptoms ( the-symptom-set )
< ::>
```



```
[ symptom-set = cluster ]
[ multiple-hard-fault-analysis = started ]
[ diagnosis-set = empty ]
[ multiple-hard-fault :: execute ( multiple-hard-fault ) ]
[ mf-diagnosis :: execute ( mf-diagnosis ) ] >

;
00
00 If there are more symptom sets in the symptom-set-queue
00 we need to do the next symptom set.
00
00 This is a kludge. We want to fire this rule every time through.
00 Normally, simple rules are only fired once (their semantics), to
00 get them to fire more than once we will use a for all.
MF-Control-Rule3
FOR ALL symptom-set in symptom-set-queue
  < [ lisp :: length ( symptom-set-queue ) > 1 ]
  ::>
  [ power-domain :: send-out-of-services ]
  [ symptom-set-queue = symptom-set-queue MINUS the-symptom-set ] >

;
00
00 No more symptom sets, send an end-contingency too.
00
MF-Control-Rule4
[ lisp :: length ( symptom-set-queue ) = 1 ]
::>
[ power-domain :: send-out-of-services ]
[ power-domain :: end-contingency ]
[ symptom-set-queue = empty ]

;
00
00 There is no termination condition on this rule group.
00 This rule group continually runs watching for symptom sets.
00

DONE
```

Multiple Hard Fault Rule Group

```
00
00 The Hard Fault Rule Group for Multiple Faults
00
```

RULE-GROUP : Multiple-Hard-Fault

```
00
00
```

```
CONTROL : ((start (MF-rule1))
(MF-rule1 (MF-rule2.1 MF-rule2.2 MF-rule2.3 MF-rule2.4
MF-rule2.5 MF-rule3.1 MF-rule3.2 MF-rule4.1
MF-rule4.2 MF-rule4.3 MF-rule4.4 MF-rule4.5))
(MF-rule3.1 (MF-rule5 MF-rule5.1))
(MF-rule3.2 (MF-rule3.2.1 MF-rule3.2.2 MF-rule3.2.3 MF-rule3.2.4
MF-rule3.2.5 MF-rule3.2.6 MF-rule3.2.7))
(MF-rule4.1 (MF-rule5 MF-rule5.1))
(MF-rule4.2 (MF-rule5 MF-rule5.1))
(MF-rule4.3 (MF-rule5 MF-rule5.1))
(MF-rule4.4 (MF-rule5 MF-rule5.1))
(MF-rule4.5 (MF-rule5 MF-rule5.1))
(MF-rule5.1 (MF-rule6.1))
(MF-rule6.1 (MF-rule6.2 MF-rule6.3))
(MF-rule6.3 (MF-rule6.4 MF-rule6.5 MF-rule6.6 MF-rule6.7
MF-rule6.8))
(MF-rule6.8 (MF-rule6.8.1 MF-rule6.8.2))
(MF-rule6.8.2 (MF-rule6.9))
(MF-rule6.9 (MF-rule6.10 MF-rule6.11))
(MF-rule6.10 (MF-rule6.12 MF-rule6.16))
(MF-rule6.12 (MF-rule6.13 MF-rule6.14 MF-rule6.15))
(MF-rule6.16 (MF-rule6.17 MF-rule6.22))
(MF-rule6.17 (MF-rule6.18))
(MF-rule6.18 (MF-rule6.19 MF-rule6.20 MF-rule6.21))
(MF-rule6.21 (MF-rule6.9))
(MF-rule6.22 (MF-rule6.23 MF-rule6.24 MF-rule6.25))
(MF-rule6.2 (MF-rule5 MF-rule5.1))
(MF-rule6.4 (MF-rule5 MF-rule5.1))
(MF-rule6.5 (MF-rule5 MF-rule5.1))
(MF-rule6.6 (MF-rule5 MF-rule5.1))
(MF-rule6.7 (MF-rule5 MF-rule5.1))
(MF-rule6.8.1 (MF-rule5 MF-rule5.1))
(MF-rule6.11 (MF-rule5 MF-rule5.1))
(MF-rule6.13 (MF-rule5 MF-rule5.1))
(MF-rule6.14 (MF-rule5 MF-rule5.1))
(MF-rule6.15 (MF-rule5 MF-rule5.1))
(MF-rule6.19 (MF-rule5 MF-rule5.1))
(MF-rule6.20 (MF-rule5 MF-rule5.1))
(MF-rule6.23 (MF-rule5 MF-rule5.1))
(MF-rule6.24 (MF-rule5 MF-rule5.1))
(MF-rule6.25 (MF-rule5 MF-rule5.1)))
```

00

00 First we need the top-symptoms of the current symptom-set.
00 This will give us an initial indication of a multiple fault
00 situation or no power to bus situation.

00

MF-rule1

```

::>
[ top-symptoms = power-domain :: top-symptoms ( symptom-set ) ]
;
00
00 Ok, things start getting quite a bit more complex here.
00 The series 2, 3, and 4 rules are all nondeterministic with respect to
00 each other. Therefore, we need to have negations of each of the
00 the series WRT to the other series in the rules themselves. This is
00 nothing new. It is a gotcha if you don't remember these things (like
00 I didn't - JDR). As of 4/26/90 we are embedding the negations of parts
00 of the series two rules into the series four rules. We don't need
00 to worry about the series three rules for a while as they depend upon
00 knowledge of activities at the lower switches. We don't yet have this
00 knowledge, therefore they will not be activated.
00 It turns out that only rule 4.5 seems to need modification.
00
00
00 The series 2.x rules deal with an under-voltage fault where there
00 is not current trip above the highest symptom; that is, the highest
00 symptom is either a single under voltage or a set of under voltages.
00
00 The 2.x rules determine broken input and output cables of switches
00 as well as possible no power to the bus situations.
00
00 All the 2.x rules first make sure that all the other switches on the
00 same bus that were on also tripped on under voltage and not on any
00 other fault.
00
00 May want to rethink these a bit. Switches do not have voltage.
00 Rule 2.5 will not fire.

00
00 Rule 2.1 checks if the top-sensor of the bus (either am0 or hm0) is
00 reading less than nominal voltage. If it is, then we probably don't
00 have power to the bus.
00
00 The rest of the 2.x rules (2.2 2.3 2.4 2.5) are cases
00 where there does appear to nominal voltage at the top-sensor, and
00 therefore have to do with cases where there may be broken switches
00 and faulty sensors.
00
MF-rule2.1
[ lisp :: length ( top-symptoms ) >= 1 ]
[ FOR ALL symptom in top-symptoms
  < [ fault of symptom = under-voltage ] > ]
[ THERE EXISTS symptom in top-symptoms
  < [ voltage of top-sensor of switch of symptom <=

```

```

        under-voltage-value of switch of symptom ] > ]
[ THERE EXISTS symptom1 in top-symptoms
  < [ FOR ALL switch1 in siblings of switch of symptom1
    WHERE [ event of switch1 <> :unknown ]
      [ command of event of switch1 = n ]
    < [ THERE EXISTS symptom in top-symptoms
      < [ switch of symptom = switch1 ] > ] > ] > ]
::>
[ diagnosis = no-power-to-bus ]
[ multiple-hard-fault-analysis = done ]
;
00
00 Rule 2.2 checks if there is nominal voltage at the sensor above one
00 of the tripped switches. If there is, then either the cable is
00 broken between that sensor and the switches (for more than one switch)
00 or the switch-input of the tripped switches is broken.
00
MF-rule2.2
[ lisp :: length ( top-symptoms ) >= 1 ]
[ FOR ALL symptom in top-symptoms
  < [ fault of symptom = under-voltage ] > ]
[ THERE EXISTS symptom1 in top-symptoms
  < [ FOR ALL switch1 in siblings of switch of symptom1
    WHERE [ event of switch1 <> :unknown ]
      [ command of event of switch1 = n ]
    < [ THERE EXISTS symptom in top-symptoms
      < [ switch of symptom = switch1 ] > ] > ] > ]
[ THERE EXISTS symptom in top-symptoms
  < [ voltage of top-sensor of switch of symptom >
    under-voltage-value of switch of symptom ]
    [ voltage of sensor-above of switch of symptom >
    under-voltage-value of switch of symptom ] > ]
::>
[ diagnosis = broken-cable-between-sensor-above-and-u-v-switches ]
[ multiple-hard-fault-analysis = done ]
;
00
00 Rule 2.3 checks the case where the sensor above the tripped switches
00 is reading less than nominal voltage and the switch above the
00 tripped switches can trip on under voltage but is reading a
00 a nominal voltage. In this case a broken output cable of the
00 switch above the tripped switches is hypothesized.
00
MF-rule2.3
[ lisp :: length ( top-symptoms ) >= 1 ]
[ FOR ALL symptom in top-symptoms
  < [ fault of symptom = under-voltage ] > ]
[ THERE EXISTS symptom1 in top-symptoms

```

```

< [ FOR ALL switch1 in siblings of switch of symptom1
  WHERE [ event of switch1 <> :unknown ]
        [ command of event of switch1 = n ]
  < [ THERE EXISTS symptom in top-symptoms
    < [ switch of symptom = switch1 ] > ] > ] > ]
[ THERE EXISTS symptom in top-symptoms
  < [ voltage of top-sensor of switch of symptom >
    under-voltage-value of switch of symptom ]
    [ voltage of sensor-above of switch of symptom <=
      under-voltage-value of switch of symptom ]
    [ under-voltage-trippable of switch-above of switch of symptom =
      true ] > ]
::>
[ diagnosis = broke-output-cable-of-switch-above ]
[ multiple-hard-fault-analysis = done ]
;
00
00 Rule 2.4 checks that the sensor above the tripped switches is reading
00 less than nominal voltage and that the switch above the tripped switches
00 is NOT trippable on under voltage and, at the same time, that the
00 sensor above the switch above the tripped switches IS reading a
00 nominal voltage. If this is the case we can hypothesize that the
00 switch above the tripped switches may have a broken input or output
00 cable or it may simply be busted.
00
MF-rule2.4
[ lisp :: length ( top-symptoms ) >= 1 ]
[ FOR ALL symptom in top-symptoms
  < [ fault of symptom = under-voltage ] > ]
[ THERE EXISTS symptom1 in top-symptoms
  < [ FOR ALL switch1 in siblings of switch of symptom1
    WHERE [ event of switch1 <> :unknown ]
          [ command of event of switch1 = n ]
    < [ THERE EXISTS symptom in top-symptoms
      < [ switch of symptom = switch1 ] > ] > ] > ]
[ THERE EXISTS symptom in top-symptoms
  < [ voltage of top-sensor of switch of symptom >
    under-voltage-value of switch of symptom ]
    [ voltage of sensor-above of switch of symptom <=
      under-voltage-value of switch of symptom ]
    [ under-voltage-trippable of switch-above of switch of symptom = false ]
    [ voltage of sensor-above of switch-above of switch of symptom >
      under-voltage-value of switch of symptom ] > ]
::>
[ diagnosis = broke-input-cable-of-switch-above ]
[ multiple-hard-fault-analysis = done ]
;
00

```

00 Rule 2.5 checks that there is a nominal voltage reading at the
00 sensor above the tripped switches and that the switch above the
00 tripped switches can trip on under voltage but is also reading
00 a nominal voltage, but the sensor above the switch above is
00 reading less than nominal voltage. In this case we can hypothesize
00 that the input cable to the switch above may be broken as well as
00 the under voltage sensor of the switch above.

00

MF-rule2.5

```
[ lisp :: length ( top-symptoms ) >= 1 ]
[ FOR ALL symptom in top-symptoms
  < [ fault of symptom = under-voltage ] > ]
[ THERE EXISTS symptom1 in top-symptoms
  < [ FOR ALL switch1 in siblings of switch of symptom1
    WHERE [ event of switch1 <> :unknown ]
      [ command of event of switch1 = n ]
    < [ THERE EXISTS symptom in top-symptoms
      < [ switch of symptom = switch1 ] > ] > ] > ]
[ THERE EXISTS symptom in top-symptoms
  < [ voltage of top-sensor of switch of symptom >
    under-voltage-value of switch of symptom ]
    [ voltage of sensor-above of switch of symptom <=
      under-voltage-value of switch of symptom ]
    [ under-voltage-trippable of switch-above of switch of symptom = true ]
    [ voltage of switch-above of switch of symptom >
      under-voltage-value of switch of symptom ]
    [ voltage of sensor-above of switch-above of switch of symptom <=
      under-voltage-value of switch of symptom ] > ]
::>
[ diagnosis =
  break-in-cable-above-switch-above-and-bad-u-v-sensor-switch-above ]
[ multiple-hard-fault-analysis = done ]
;
```

00

00 The 3.x.x series of rules deal with two special cases.
00 3.1 checks that there is more than one top-symptom where all
00 the top-symptoms are at the bottom level of switches (right above
00 the loads, and that all the top-symptoms tripped on either
00 over-current or fast-trip. Finally all the top-symptoms are related
00 by an activity that is using the switches of the top-symptoms.
00 3.2 checks that the set of top-symptoms are not related by the same
00 activity and that the switches are at the bottom level and
00 that they all tripped on fast-trip.

00

00

00 Rule 3.1, as stated above, checks that the switches of all the

00 symptoms are at the bottom level and tripped on either over-current
00 or fast-trip. The switches of the symptoms are also related by
00 being used by the same activity.
00 IF these conditions are satisfied then perhaps the reason this set
00 of symptoms occurred is that the activity is behaving badly. To
00 test this the 3.1.x rules will first flip and then close the switches
00 in an effort to see if there is any repeatability in the symptoms.
00

MF-rule3.1

```
[ lisp :: length ( top-symptoms ) > 1 ]  
[ FOR ALL symptom in top-symptoms  
  < [ switches-below of switch of symptom = empty ] > ]  
[ FOR ALL symptom in top-symptoms  
  < [ fault of symptom = over-current ]  
    OR  
    [ fault of symptom = fast-trip ] > ]  
[ THERE EXISTS symptom in top-symptoms  
  < [ FOR ALL symptom1 in top-symptoms  
    < [ activity of switch of symptom =  
      activity of switch of symptom1 ] > ] > ]  
::>  
[ the-individual-symptoms = top-symptoms ]  
0 until we actually have activities to look at on the workstation  
0 we will treat these as individual cases.  
0[ switches-to-test = empty ]
```

;
00

00 This is where the 3.1.x rules will go
00

00
00 Rule 3.2 checks that all the symptoms are fast-trip and all at the
00 bottom level. Also all the switches of the symptoms are not related
00 by being used by the same activity. In this case we consider the
00 possibility that there was a fault below one of the switches and
00 the other switches also tripped on fast trip due to energy storage
00 in their loads.
00 To test this we flip all the switches and try to get ONE of them
00 to retrip on fast-trip.
00

MF-rule3.2

```
[ lisp :: length ( top-symptoms ) > 1 ]  
[ FOR ALL symptom in top-symptoms  
  < [ switches-below of switch of symptom = empty ]  
    [ fault of symptom = fast-trip ] > ]  
[ THERE EXISTS symptom in top-symptoms  
  < [ THERE EXISTS symptom1 in top-symptoms  
    < [ activity of switch of symptom <>
```

```

        activity of switch of symptom1 ] > ] > ]
::>
[ switches-to-test = empty ]
;
00
00 Rule 3.2.1 grabs all the switches that may be tested according
00 to the permission-to-test field of the scheduled event for the switch.
00
MF-rule3.2.1
::>
[ FOR ALL symptom in top-symptoms
  WHERE [ permission-to-test of event of switch = y ]
    < [ switches-to-test = switches-to-test PLUS switch of symptom ] > ]
;
00
00 Rule 3.2.2. actually initiates the flipping process if there are
00 switches to test.
00
MF-rule3.2.2
[ lisp :: length ( switches-to-test ) > 0 ]
::>
[ new-symptoms = power-domain :: flip-switches ( switches-to-test ) ]
;
00
00 Rule 3.2.3 checks to see if there aren't any switches to test. If there
00 aren't then we diagnose that situation (basically we can't determine
00 anything and can only notify the user).
00
MF-rule3.2.3
[ lisp :: length ( switches-to-test ) = 0 ]
::>
[ diagnosis = no-permission-to-test-possible-backrush ]
;
00
00 Rule 3.2.4 checks if we got more than one symptom as a result of
00 flipping the switches. If so we have a problem. Flipping flips
00 the switches one at a time individually, there should only be one
00 symptom as a result of this operation.
00
MF-rule3.2.4
[ lisp :: length ( new-symptoms ) > 1 ]
::>
[ diagnosis = unexpected-to-many-retrips-possible-backrush ]
;
00
00 Rule 3.2.5 checks to see if we didn't get any symptoms as a result
00 of flipping these switches. If that is the case we also don't
00 know what the problem is. If there is a short below a switch it ought

```



```
00 to retrip the switch when the switch gets turned on. Could be a
00 misbehaving load I suppose.
00 This diagnosis should check if there were some switches that didn't
00 have permission to test. That is, are switches-to-test and top-symptoms
00 of the same length?
```

```
00
00 MF-rule3.2.5
00 [ lisp :: length ( new-symptoms ) = 0 ]
00 ::>
00 [ diagnosis = no-retrips-on-flips-possible-backrush ]
```

```
00 ;
00
00 Rule 3.2.6 checks that if we did get a single new symptom and
00 it is also a fast trip and one of the same switches of the original
00 top-symptoms, then we can diagnose this situation as a possible
00 backrush situation.
```

```
00
00 MF-rule3.2.6
00 [ lisp :: length ( new-symptoms ) = 1 ]
00 [ THERE EXISTS symptom in new-symptoms
00   WHERE [ fault of symptom = fast-trip ]
00   < [ THERE EXISTS symptom1 in top-symptoms
00     < [ switch of symptom = switch of symptom1 ] > ] > ]
00 ::>
00 [ diagnosis = found-possible-backrush ]
```

```
00 ;
00
00 Rule 3.2.7 checks that if we did get a single new symptom but it
00 isn't a fast trip or it isn't one of the switches of the original
00 top-symptoms, then we have another unexpected situation.
```

```
00
00 MF-rule3.2.7
00 [ lisp :: length ( new-symptoms ) = 1 ]
00 [ THERE EXISTS symptom in new-symptoms
00   < [ fault of symptom <> fast-trip ]
00   OR
00   [ FOR ALL symptom1 in top-symptoms
00     < [ switch of symptom <> switch of symptom1 ] > ] > ]
00 ::>
00 [ diagnosis = unexpected-retrip-possible-backrush ]
```

```
00 ;
00
00 The 4.x rules set things up to treat multiple top symptoms as
00 individual top symptoms, as different faults. It grabs
00 all the cases that weren't applicable in the 2.x and 3.x series.
```

```
00
00 MF-rule4.1
```

```
[ lisp :: length ( top-symptoms ) > 1 ]
[ THERE EXISTS symptom in top-symptoms
  < [ switches-below of switch of symptom = empty ]
    [ fault of symptom = over-current ]
    [ THERE EXISTS symptom1 in top-symptoms
      WHERE [ symptom <> symptom1 ]
        < [ activity of switch of symptom <>
          activity of switch of symptom1 ] > ] > ]
::>
[ the-individual-symptoms = symptom-set ]
;
MF-rule4.2
[ lisp :: length ( top-symptoms ) > 1 ]
[ FOR ALL symptom in top-symptoms
  < [ fault of symptom = under-voltage ] > ]
[ THERE EXISTS symptom in top-symptoms
  < [ THERE EXISTS switch1 in siblings of switch of symptom
    < [ command of event of switch1 = n ]
      [ FOR ALL symptom1 in top-symptoms
        < [ switch1 <> switch of symptom1 ] > ] > ] > ]
::>
[ the-individual-symptoms = top-symptoms ]
;
MF-rule4.3
[ lisp :: length ( top-symptoms ) > 1 ]
[ FOR ALL symptom in top-symptoms
  < [ fault of symptom <> under-voltage ] > ]
::>
[ the-individual-symptoms = top-symptoms ]
;
MF-rule4.4
[ lisp :: length ( top-symptoms ) > 1 ]
[ THERE EXISTS symptom in top-symptoms
  < [ fault of symptom = under-voltage ] > ]
[ THERE EXISTS symptom in top-symptoms
  < [ fault of symptom <> under-voltage ] > ]
::>
[ the-individual-symptoms = top-symptoms ]
;
MF-rule4.5
[ lisp :: length ( top-symptoms ) = 1 ]
@@ This next selector is for the negation of the series two rules (Bah!)
@@ The under voltage should have been caught by the series two rules.
[ THERE EXISTS symptom in top-symptoms
  < [ fault of symptom <> under-voltage ] > ]
::>
[ the-individual-symptoms = top-symptoms ]
;
```

00

00 There are two ways that we can finish diagnosing hard faults.
00 One way is by concluding a value for diagnosis based on one of the cases
00 in the 2.x and 3.x series of rules. The other is by going through all
00 the individual top symptoms and getting a diagnosis for all of them
00 (adding them to the diagnosis set one by one).
00 The 5.x series of rules are simply to manage the looping necessary
00 to diagnose each top-symptom individually as individual faults.

00

```
MF-rule5
[ the-individual-symptoms = empty ]
::>
[ multiple-hard-fault-analysis = done ]
;
MF-rule5.1
[ the-individual-symptoms <> empty ]
::>
[ top-symptom = lisp :: first ( the-individual-symptoms ) ]
[ the-individual-symptoms = lisp :: rest ( the-individual-symptoms ) ]
;
```

00

00 The 6.x series of rules are used to diagnose single faults with
00 only a single top symptom (not multiple tops, they were handled
00 above). This is very similar, but simpler, to the original set
00 of rules for doing hard-fault diagnosis. Some simplicity is
00 obtained by handling backrush and under-voltage faults in the
00 earlier rules.

00

00 The types of faults that will be detected in the 6.x series of rules
00 are over current and fast trip faults. They can also be with
00 a number of masked faults where sensors didn't work properly.
00 Additionally, these rules are smart enough that if a fault is
00 found that is lower than the top, and there are other potential
00 faults in a different portion of the tree below the top, then
00 those other faults are added to the-individual-symptoms so that
00 they may be diagnosed as well.

00

00

00 Rule 6.1 simply opens all the switches from the top-symptom on
00 down. This is an initialization step.

00

```
MF-rule6.1
::>
[ switch of top-symptom = power-domain :: kludge-switch ( top-symptom ) ]
[ fault of top-symptom = power-domain :: kludge-fault ( top-symptom ) ]
[ new-symptoms = power-domain :: open-switches ( top-symptom ) ]
```

```

;
00
00 Rule 6.2 determines if we got some trips as a result of opening
00 the switches. Getting new symptoms during the open operations is an
00 unexpected result and notification to the user will be given of
00 this problem.
00
MF-rule6.2
[ lisp :: length ( new-symptoms ) > 0 ]
::>
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis ( :name unexpected-symptoms-during-open
                                   :top-symp top-symptom
                                   :slot1 new-symptoms ) ]

;
00
00 Rule 6.3 flips the top switch. It collects any symptoms that may arise
00 as a result of the flip.
00
MF-rule6.3
[ lisp :: length ( new-symptoms ) = 0 ]
::>
[ new-symptoms = power-domain :: flip-switch ( switch of top-symptom ) ]

;
00
00 Rule 6.4 checks if there was more than one symptom as a result of the
00 flip of the single top switch. If so this is an unexpected situation
00 and the user will be notified.
00
MF-rule6.4
[ lisp :: length ( new-symptoms ) > 1 ]
::>
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis
    ( :name unexpected-too-many-symptoms-flip-top
      :top-symp top-symptom :slot1 new-symptoms ) ]

;
00
00 Rule 6.5 finds the case where the top switch retrips on the
00 same trip as a result of the flip operation. This is a strong
00 indication of a short directly below the top switch.
00 The diagnosis also will note if there were any fast trips at
00 the bottom level (and that the top switch also tripped on fast trip)
00 and suggest the possibility of energy storage in the loads
00 driving those bottom fast trips.
00
MF-rule6.5
[ lisp :: length ( new-symptoms ) = 1 ]

```

```
[ THERE EXISTS symptom in new-symptoms
  < [ switch of symptom = switch of top-symptom ]
    [ fault of symptom = fault of top-symptom ] > ]
::>
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis ( :name retrip-on-flip
                                   :top-symp top-symptom ) ]
;
00
00 Rule 6.6 also detects a single retrip on the flip operation, but
00 this retrip is either a different symptom or a different switch or
00 both, and therefore is unexpected. The user will be notified.
00
MF-rule6.6
[ lisp :: length ( new-symptoms ) = 1 ]
[ THERE EXISTS symptom in new-symptoms
  < [ fault of symptom <> fault of top-symptom ]
    OR
    [ switch of symptom <> switch of top-symptom ] > ]
::>
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis ( :name unexpected-retrip-during-flip
                                   :top-symp top-symptom
                                   :slot1 new-symptoms ) ]
;
00
00 Rule 6.7 checks the case where the top switch did not retrip during
00 the flip operation and notes that there are no switches below
00 the top switch. In this case, the fault is not found and there
00 is nowhere else to check. Perhaps the fault burned itself clear?
00
MF-rule6.7
[ lisp :: length ( new-symptoms ) = 0 ]
[ switches-below of switch of top-symptom = empty ]
::>
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis ( :name not-found-no-levels
                                   :top-symp top-symptom ) ]
;
00
00 Rule 6.8 checks that there were no new trips during the flip operation
00 on the top switch, but in this case there are switches below that
00 can be tested too. So this rule initializes some variables to
00 start flipping and closing the lower switches.
00 This rule also closes the top switch in preparation for flipping
00 and closing the lower switches
00
00 It is at this point of the game that we are now interested in
```

00 finding masked faults. It is a possibility that
00 a lower switch's current trip sensor is broken and that therefore
00 the top switch tripped.

00

MF-rule6.8

```
[ lisp :: length ( new-symptoms ) = 0 ]  
[ lisp :: length ( switches-below of switch of top-symptom ) > 0 ]  
::>  
[ switches-below-to-test = empty ]  
[ switches-below-cant-test = empty ]  
[ switches-below = switches-below of switch of top-symptom ]  
[ new-symptoms = power-domain :: close-switch ( switch of top-symptom ) ]  
;
```

00

00 Rule 6.8.1 checks to see if we got new symptoms during the close of
00 of the top switch. If we did this is completely unexpected.
00 A new symptom should have only come during the flip (since we
00 are diagnosing current trips right now). We will notify
00 the user of this problem.

00

MF-rule6.8.1

```
[ lisp :: length ( new-symptoms ) > 0 ]  
::>  
[ diagnosis-set = diagnosis-set PLUS  
  power-domain :: make-diagnosis ( :name unexpected-trips-during-close-top  
    :top-symp top-symptom  
    :slot1 new-symptoms ) ]  
;
```

00

00 Rule 6.8.2 checks that we didn't get any new symptoms as a result
00 of the close of the top switch and therefore we can go on to rule
00 6.9

00

MF-rule6.8.2

```
[ lisp :: length ( new-symptoms ) = 0 ]  
::>  
[ level = 1 ]  
;
```

00

00 Rule 6.9 sets up the switches-below-to-test and switches-below-cant-test
00 variables based upon the status of the switch below and whether it
00 was supposed to be on.

00

MF-rule6.9

```
[ level > 0 ]  
::>  
[ FOR ALL switch in switches-below
```

```

WHERE [ tripped of switch = under-voltage ]
      [ command of event of switch = n ]
      [ permission-to-test of event of switch = y ]
< [ switches-below-to-test = switches-below-to-test PLUS switch ] > ]
[ FOR ALL switch in switches-below
  WHERE [ command of event of switch = n ]
        [ permission-to-test of event of switch = n ]
        OR
        [ command of event of switch = n ]
        [ tripped of switch <> under-voltage ]
    < [ switches-below-cant-test = switches-below-cant-test PLUS switch ] > ]
;
%%
%% Rule 6.10 flips the switches that can be tested to see if
%% we get any retrips.
%%
MF-rule6.10
[ lisp :: length ( switches-below-to-test ) > 0 ]
::>
[ new-symptoms = power-domain :: flip-switches ( switches-below-to-test ) ]
;
%%
%% Rule 6.11 checks for the possibility that none of the lower switches
%% may be tested. In this case we have to diagnose with what we got.
%%
MF-rule6.11
[ lisp :: length ( switches-below-to-test ) = 0 ]
::>
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis ( :name not-found-cant-test-further
                                   :top-symp top-symptom
                                   :slot1 switches-below-cant-test ) ]
;
%%
%% Rule 6.12 checks to see if we got more than one new symptom during
%% the flips of the lower switches. This is possible if we have a
%% masked fault. At this point we find the top symptoms of the new
%% symptoms.
%%
MF-rule6.12
[ lisp :: length ( new-symptoms ) > 1 ]
::>
[ new-top-symptoms = power-domain :: top-symptoms ( new-symptoms ) ]
;
%%
%% Rule 6.13 checks if we have more than one new top symptom.
%% If we do, this is unexpected and the user will have to be notified.
%%

```

MF-rule6.13

```
[ lisp :: length ( new-top-symptoms ) > 1 ]
::>
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis ( :name unexpected-to-many-tops-after-flips
                                   :top-symp top-symptom
                                   :slot1 new-symptoms ) ]
```

;

00

00 Rule 6.14 checks that we did have only one new top symptom and that
00 it turns out to be the same as the original top symptom. This looks
00 like a case where we found a masked fault.
00 We also reclose all the switches above the culprit switch to enable
00 further testing of other possible faults below the top switch.
00 We determine if there are any other possible faults by calling
00 the function new-diagnosable-symptoms that looks for other fast
00 trips or current trips below the top switch and either siblings
00 of the switch that was the culprit or below the siblings of said
00 switch.

00

MF-rule6.14

```
[ lisp :: length ( new-top-symptoms ) = 1 ]
[ THERE EXISTS symptom in new-top-symptoms
  < [ fault of symptom = fault of top-symptom ]
    [ switch of symptom = switch of top-symptom ] > ]
::>
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis ( :name found-below
                                   :top-symp top-symptom
                                   :slot1 which-switch ) ]

[ power-domain :: reclose-switches ( top-symptom which-switch ) ]
[ FOR ALL symptom in
  power-domain :: new-diagnosable-symptoms ( top-symptom which-switch )
  < [ the-individual-symptoms = the-individual-symptoms PLUS symptom ] > ]
```

;

00

00 Rule 6.15 also checks that we got one new top symptom during the flips
00 of the lower switches. But in this case it is not the same as the
00 original top symptom. This is an unexpected situation and the user
00 will be notified.

00

MF-rule6.15

```
[ lisp :: length ( new-top-symptoms ) = 1 ]
[ THERE EXISTS symptom in new-top-symptoms
  < [ fault of symptom <> fault of top-symptom ]
    OR
    [ switch of symptom <> switch of top-symptom ] > ]
::>
```



```
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis ( :name unexpected-different-top-after-flips
                                   :top-symp top-symptom
                                   :slot1 new-symptoms ) ]
;

00
00 Rule 6.16 checks that we didn't get any new symptoms and therefore
00 we close the lower switches now.
00
MF-rule6.16
[ lisp :: length ( new-symptoms ) = 0 ]
::>
[ new-symptoms = power-domain :: close-switches ( switches-below-to-test ) ]
;
00
00 Rule 6.17 checks that we didn't get any new symptoms as a result of
00 closing the lower switches. If this is the case we determine that
00 we haven't yet found the fault at this level of testing. We set
00 switches-below to empty in preparation for the next level of testing.
00
MF-rule6.17
[ lisp :: length ( new-symptoms ) = 0 ]
::>
[ switches-below = empty ]
;
00
00 Rule 6.18 collects the next set of lower switches that might
00 be testable.
00
MF-rule6.18
::>
[ FOR ALL switch in switches-below-to-test
  < [ switches-below = switches-below UNION switches-below of switch ] > ]
;
00
00 Rule 6.19 checks if there are no lower switches below to test.
00 If this is the case and if there were switches below the top switch
00 that we could not test we can diagnose the case as a not found case
00 in light of the fact that we also couldn't test some of the switches.
00
MF-rule6.19
[ lisp :: length ( switches-below ) = 0 ]
[ lisp :: length ( switches-below-cant-test ) > 0 ]
::>
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis ( :name not-found-cant-test-further
                                   :top-symp top-symptom
```

```

                                :slot1 switches-below-cant-test ) ]
;
00
00 Rule 6.20 checks that there are no lower switches below to test and
00 that we have tested everything that could be tested. This is a simply
00 not found case. Perhaps it was a transient?
00
MF-rule6.20
[ lisp :: length ( switches-below ) = 0 ]
[ lisp :: length ( switches-below-cant-test ) = 0 ]
::>
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis ( :name not-found-all-tested
                                   :top-symp top-symptom ) ]
;
00
00 Rule 6.21 checks that there are lower switches that might be testable
00 and therefore sets up the variables so that we can go back to rule 6.9
00 and continue testing at the next level.
00
MF-rule6.21
[ lisp :: length ( switches-below ) > 0 ]
::>
[ switches-below-to-test = empty ]
[ level = level PLUS 1 ]
;
00
00 Rule 6.22 determines that there were new symptoms as a result of
00 the close of the lower switches. We simply get the top symptoms
00 of the new symptoms in this rule.
MF-rule6.22
[ lisp :: length ( new-symptoms ) > 0 ]
::>
[ new-top-symptoms = power-domain :: top-symptoms ( new-symptoms ) ]
;
00
00 Rule 6.23 notes that we got a single top symptom and it is the
00 same as the original top symptom. The difference between this
00 rule and 6.14 is that this happened during the closes. The
00 diagnosis needs to look at the topology to determine what might
00 account for this (namely and over current trip and multiple
00 lower switches drawing too much current in tandem while intermediate
00 switches may have broken sensors or be the same rating of the
00 top switch). If this is a fast trip case it doesn't make much
00 sense.
00 We don't do reclosing here like we do in 6.14 since this
00 situation is not as well defined.
00

```

MF-rule6.23

```
[ lisp :: length ( new-top-symptoms ) = 1 ]
[ THERE EXISTS symptom in new-top-symptoms
  < [ fault of symptom = fault of top-symptom ]
    [ switch of symptom = switch of top-symptom ] > ]
::>
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis ( :name possible-found
                                   :top-symp top-symptom
                                   :slot1 which-switch
                                   :slot2 switches-below-to-test ) ]
```

```
;
;;
;; Rule 6.24 checks that we got one new top symptom as a result of the
;; closes of the lower switches but that this new symptom isn't the
;; same as the original top symptom. This is an unexpected situation
;; and the user will be notified.
;;
```

MF-rule6.24

```
[ lisp :: length ( new-top-symptoms ) = 1 ]
[ THERE EXISTS symptom in new-top-symptoms
  < [ fault of symptom <> fault of top-symptom ]
    OR
    [ switch of symptom <> switch of top-symptom ] > ]
::>
```

```
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis
    ( :name unexpected-different-trip-during-closes
      :top-symp top-symptom :slot1 new-symptoms ) ]
```

```
;
;;
;; Rule 6.25 detects the case where more than one new top symptom
;; resulted from the close operation of the lower switches. This is
;; also unexpected and the user will be notified.
;;
```

MF-rule6.25

```
[ lisp :: length ( new-top-symptoms ) > 1 ]
::>
[ diagnosis-set = diagnosis-set PLUS
  power-domain :: make-diagnosis ( :name unexpected-new-trips-during-closes
                                   :top-symp top-symptom
                                   :slot1 new-symptoms ) ]
```

```
;;
;; The termination condition is setup by rule 5
;;
;
[ multiple-hard-fault-analysis = done ]
```

DONE

Diagnosis Rule Group

```
00
00 The Diagnosis Rule Group
00
00 This rule group takes a diagnosis and prints out relevant information about
00 the diagnosis and sets up any out of service information on switches.
00

RULE-GROUP : MF-Diagnosis

00
00 MF-diag-1 (no-power-to-bus)
00   Diagnosed in MF-rule2.1
00
00   MF-diag-1
00     [ diagnosis = no-power-to-bus ]
00     ::>
00     [ power-domain :: write ( "The following switches tripped on under voltage:" ) ]
00     [ FOR ALL symptom in top-symptoms
00       < [ power-domain :: write ( "  "a" switch of symptom ) ] > ]
00     [ THERE EXISTS symptom in top-symptoms
00       < [ power-domain :: write
00         ( "Sensor "a, the top sensor of the bus, registers less than the nominal"
00           top-sensor of switch of symptom ) ]
00         [ power-domain :: write ( "amount of voltage." ) ] > ]
00     [ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
00     [ power-domain :: write ( " Most Likely:" ) ]
00     [ power-domain :: write ( " Less than nominal voltage supplied to bus." ) ]
00     [ FOR ALL symptom in top-symptoms
00       < [ power-domain :: out-of-service ( switch of symptom ) ] > ]
00     [ diagnosis = :unknown ]
00   ;
00
00 MF-diag-2 (broken-cable-between-sensor-above-and-u-v-switches)
00   Diagnosed in MF-rule2.2
00
00   MF-diag-2
00     [ diagnosis = broken-cable-between-sensor-above-and-u-v-switches ]
00     ::>
00     [ power-domain :: write ( "The following switches tripped on under voltage:" ) ]
00     [ FOR ALL symptom in top-symptoms
00       < [ power-domain :: write ( "  "a" switch of symptom ) ] > ]
00     [ THERE EXISTS symptom in top-symptoms
00       < [ power-domain :: write
00         ( "Sensor "a, the top sensor of the bus, registers less nominal voltage."
00           top-sensor of switch of symptom ) ]
00         [ power-domain :: write ( "amount of voltage." ) ] > ]
00     [ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
00     [ power-domain :: write ( " Most Likely:" ) ]
00     [ power-domain :: write ( " Less than nominal voltage supplied to bus." ) ]
00     [ FOR ALL symptom in top-symptoms
00       < [ power-domain :: out-of-service ( switch of symptom ) ] > ]
00     [ diagnosis = :unknown ]
00   ;
```

```

        top-sensor of switch of symptom ) ] > ]
[ THERE EXISTS symptom in top-symptoms
  < [ power-domain :: write
    ( "Sensor "a, the sensor above the tripped switches, also registers"
      sensor-above of switch of symptom ) ]
    [ power-domain :: write ( "nominal voltage." ) ] > ]
[ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
[ power-domain :: write ( " Most Likely:" ) ]
[ power-domain :: write
  ( " Switch input cables to the switches disconnected in some fashion." ) ]
[ power-domain :: write ( " Less Likely:" ) ]
[ power-domain :: write
  ( " Break in cable between the sensor above the tripped switches and the bus" ) ]
[ power-domain :: write ( "of the tripped switches." ) ]
[ FOR ALL symptom in top-symptoms
  < [ power-domain :: out-of-service ( switch of symptom ) ] > ]
[ diagnosis = :unknown ]
;
00
00 MF-diag-3 (broke-output-cable-of-switch-above)
00   Diagnosed in MF-rule2.3
00
MF-diag-3
[ diagnosis = broke-output-cable-of-switch-above ]
::>
[ power-domain :: write ( "The following switches tripped on under voltage:" ) ]
[ FOR ALL symptom in top-symptoms
  < [ power-domain :: write ( " "a" switch of symptom ) ] > ]
[ THERE EXISTS symptom in top-symptoms
  < [ power-domain :: write
    ( "Sensor "a, the top sensor of the bus, registers nominal voltage."
      top-sensor of switch of symptom ) ] > ]
[ THERE EXISTS symptom in top-symptoms
  < [ power-domain :: write
    ( "Sensor "a, the sensor above the tripped switches, registers less"
      sensor-above of switch of symptom ) ]
    [ power-domain :: write ( "than nominal voltage" ) ]
    [ power-domain :: write
      ( "while, "a, the switch above the tripped switches, registers a nominal voltage."
        switch-above of switch of symptom ) ] > ]
[ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
[ power-domain :: write ( " Most Likely:" ) ]
[ power-domain :: write
  ( " The switch output cable of the switch above the tripped switches has" ) ]
[ power-domain :: write ( "been disconnected in some fashion." ) ]
[ THERE EXISTS symptom in top-symptoms
  < [ power-domain :: out-of-service
    ( switch-above of switch of symptom ) ] > ]

```

```

[ diagnosis = :unknown ]
;
00
00 MF-diag-4 (broke-input-cable-of-switch-above)
00   Diagnosed in MF-rule2.4
00
MF-diag-4
[ diagnosis = broke-input-cable-of-switch-above ]
::>
[ power-domain :: write ( "The following switches tripped on under voltage:" ) ]
[ FOR ALL symptom in top-symptoms
  < [ power-domain :: write ( "  "a" switch of symptom ) ] > ]
[ THERE EXISTS symptom in top-symptoms
  < [ power-domain :: write
    ( "Sensor "a, the top sensor of the bus, registers nominal voltage."
      top-sensor of switch of symptom ) ] > ]
[ THERE EXISTS symptom in top-symptoms
  < [ power-domain :: write
    ( "Sensor "a, the sensor above the tripped switches, registers less than"
      sensor-above of switch of symptom ) ]
    [ power-domain :: write ( "nominal voltage," ) ]
    [ power-domain :: write
      ( "however, "a, the sensor above the switch above the tripped switches also"
        sensor-above of switch-above of switch of symptom ) ]
    [ power-domain :: write ( "registers less than nominal voltage and" ) ]
    [ power-domain :: write
      ( "'a, the switch above the tripped switches, cannot trip on under voltage."
        switch-above of switch of symptom ) ] > ]
[ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
[ power-domain :: write ( " Most Likely:" ) ]
[ power-domain :: write
  ( " The switch input or output cable of the switch above the tripped switches" ) ]
[ power-domain :: write ( "has been disconnected in some fashion." ) ]
[ THERE EXISTS symptom in top-symptoms
  < [ power-domain :: out-of-service
    ( switch-above of switch of symptom ) ] > ]
[ diagnosis = :unknown ]
;
00
00 MF-diag-5 (break-in-cable-above-switch-above-and-bad-u-v-sensor-switch-above)
00   Diagnosed in MF-rule2.5
00
MF-diag-5
[ diagnosis =
  broke-in-cable-above-switch-above-and-bad-u-v-sensor-switch-above ]
::>
[ power-domain :: write ( "The following switches tripped on under voltage:" ) ]
[ FOR ALL symptom in top-symptoms

```

```

    < [ power-domain :: write ( "  "a" switch of symptom ) ] > ]
[ THERE EXISTS symptom in top-symptoms
    < [ power-domain :: write
        ( "Sensor "a, the top sensor of the bus, registers nominal voltage."
          top-sensor of switch of symptom ) ] > ]
[ THERE EXISTS symptom in top-symptoms
    < [ power-domain :: write
        ( "Sensor "a, the sensor above the tripped switches, registers less than"
          sensor-above of switch of symptom ) ]
    [ power-domain :: write ( "nominal voltage," ) ]
    [ power-domain :: write
        ( "however, "a, the sensor above the switch above the tripped switches also"
          sensor-above of switch-above of switch of symptom ) ]
    [ power-domain :: write ( "registers less than nominal voltage and" ) ]
    [ power-domain :: write
        ( "'a, the switch above the tripped switches, should have tripped but did not."
          switch-above of switch of symptom ) ] > ]
[ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
[ power-domain :: write ( " Most Likely:" ) ]
[ power-domain :: write
    ( " The switch input cable of the switch above the tripped switches has been" ) ]
[ power-domain :: write ( "disconnected in some fashion and" ) ]
[ power-domain :: write ( " also has a bad under voltage sensor." ) ]
[ THERE EXISTS symptom in top-symptoms
    < [ power-domain :: out-of-service
        ( switch-above of switch of symptom ) ] > ]
[ diagnosis = :unknown ]
;

00
00 MF-diag-6 (no-permission-to-test-possible-backrush)
00 Diagnosed in MF-rule3.2.3
00
MF-diag-6
[ diagnosis = no-permission-to-test-possible-backrush ]
::>
[ power-domain :: write ( "The following switches tripped on fast trip:" ) ]
[ FOR ALL symptom in top-symptoms
    < [ power-domain :: write ( "  "a" switch of symptom ) ] > ]
[ power-domain :: write ( "None of the switches has permission to test." ) ]
[ power-domain :: write
    ( "It is possible that these switches tripped on fast trip due to a low impedance" ) ]
[ power-domain :: write
    ( "short below one of them and the others due to energy storage in the loads." ) ]
[ FOR ALL symptom in top-symptoms
    < [ power-domain :: out-of-service ( switch of symptom ) ] > ]
[ diagnosis = :unknown ]
;

```

```

00
00 MF-diag-7 (unexpected-to-many-retrips-possible-backrush)
00   Diagnosed in MF-rule3.2.4
00
MF-diag-7
[ diagnosis = unexpected-to-many-retrips-possible-backrush ]
::>
[ power-domain :: write ( "The following switches tripped on fast trip:" ) ]
[ FOR ALL symptom in top-symptoms
  < [ power-domain :: write ( "  a switch of symptom ) ] > ]
[ power-domain :: write
  ( "During testing by flipping the switches the following symptoms occurred:" ) ]
[ FOR ALL symptom in new-symptoms
  < [ power-domain :: write ( "a on a switch of symptom fault of symptom ) ] > ]
[ power-domain :: write
  ( "This is not a situation that is diagnosable in the existing rule set." ) ]
[ FOR ALL symptom in top-symptoms
  < [ power-domain :: out-of-service ( switch of symptom ) ] > ]
[ diagnosis = :unknown ]
;
00
00 MF-diag-8 (no-retrips-on-flips-possible-backrush)
00   Diagnosed in MF-rule3.2.5
00
MF-diag-8
[ diagnosis = no-retrips-on-flips-possible-backrush ]
[ lisp :: length ( switches-to-test ) = lisp :: length ( top-symptoms ) ]
::>
[ power-domain :: write ( "The following switches tripped on fast trip:" ) ]
[ FOR ALL symptom in top-symptoms
  < [ power-domain :: write ( "  a switch of symptom ) ] > ]
[ power-domain :: write ( "None of the switches retripped during testing." ) ]
[ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
[ power-domain :: write ( " Most Likely:" ) ]
[ power-domain :: write ( "  A low impedance short that was burned clear." ) ]
[ power-domain :: write ( "  A transient in a load below one of the switches." ) ]
[ power-domain :: write
  ( "Cause of other switches tripping could be due to energy storage in the loads" ) ]
[ power-domain :: write ( "below them." ) ]
[ FOR ALL symptom in top-symptoms
  < [ power-domain :: out-of-service ( switch of symptom ) ] > ]
[ diagnosis = :unknown ]
;
00
00 MF-diag-9 (no-retrips-on-flips-possible-backrush)
00   Diagnosed in MF-rule3.2.5
00
MF-diag-9

```



```
[ diagnosis = no-retrips-on-flips-possible-backrush ]
[ lisp :: length ( switches-to-test ) < lisp :: length ( top-symptoms ) ]
::>
[ power-domain :: write ( "The following switches tripped on fast trip:" ) ]
[ FOR ALL symptom in top-symptoms
  < [ power-domain :: write ( "  "a" switch of symptom ) ] > ]
[ power-domain :: write ( "None of the switches retripped during testing." ) ]
[ power-domain :: write
  ( "However, only the following switches had permission to test:" ) ]
[ FOR ALL switch in switches-to-test
  < [ power-domain :: write ( "  "a" switch ) ] > ]
[ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
[ power-domain :: write ( " Most Likely:" ) ]
[ power-domain :: write
  ( " A low impedance short below one of the switches that was not tested." ) ]
[ power-domain :: write
  ( "Cause of other switches tripping could be due to energy storage in the loads" ) ]
[ power-domain :: write ( "below them." ) ]
[ FOR ALL symptom in top-symptoms
  < [ power-domain :: out-of-service ( switch of symptom ) ] > ]
[ diagnosis = :unknown ]
;
00
00 MF-diag-10 (found-possible-backrush)
00 Diagnosed in MF-rule3.2.6
00
MF-diag-10
[ diagnosis = found-possible-backrush ]
[ lisp :: length ( switches-to-test ) < lisp :: length ( top-symptoms ) ]
::>
[ power-domain :: write ( "The following switches tripped on fast trip:" ) ]
[ FOR ALL symptom in top-symptoms
  < [ power-domain :: write ( "  "a" switch of symptom ) ] > ]
[ THERE EXISTS symptom in new-symptoms
  < [ power-domain :: write
    ( "During testing of the switches, "a, retripped on fast trip."
      switch of symptom ) ] > ]
[ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
[ power-domain :: write ( " Most Likely:" ) ]
[ THERE EXISTS symptom in new-symptoms
  < [ power-domain :: write
    ( " Low impedance short below "a." switch of symptom ) ] > ]
[ power-domain :: write
  ( "Cause of other switches tripping due to energy storage in the loads below them." ) ]
[ THERE EXISTS symptom in new-symptoms
  < [ power-domain :: out-of-service ( switch of symptom ) ]
  [ FOR ALL symptom1 in top-symptoms
    WHERE [ switch of symptom1 <> switch of symptom ]
```

```

        < [ power-domain :: out-of-service ( switch of symptom1 ) ] > ] > ]
    [ diagnosis = :unknown ]
;

00
00 MF-diag-11 (unexpected-retrip-possible-backrush)
00   Diagnosed in MF-rule3.2.7
00
    MF-diag-11
    [ diagnosis = unexpected-retrip-possible-backrush ]
    ::>
    [ power-domain :: write ( "The following switches tripped on fast trip:" ) ]
    [ FOR ALL symptom in top-symptoms
      < [ power-domain :: write ( "  a" switch of symptom ) ] > ]
    [ power-domain :: write
      ( "During testing by flipping the switches the following symptoms occurred:" ) ]
    [ FOR ALL symptom in new-symptoms
      < [ power-domain :: write ( "  a on a" switch of symptom fault of symptom ) ] > ]
    [ power-domain :: write
      ( "This is not a situation that is diagnosable in the existing rule set." ) ]
    [ FOR ALL symptom in top-symptoms
      < [ power-domain :: out-of-service ( switch of symptom ) ] > ]
    [ diagnosis = :unknown ]
;

00
00 MF-diag-12 (unexpected-symptoms-during-open)
00   Diagnosed in MF-rule6.2
00
    MF-diag-12
    FOR ALL diagnosis in diagnosis-set
    WHERE [ name of diagnosis = unexpected-symptoms-during-open ]
    < ::>
    [ diagnosis-set = diagnosis-set MINUS diagnosis ]
    [ power-domain :: write ( "a tripped on a."
      switch of top-symp of diagnosis
      fault of top-symp of diagnosis ) ]
    [ power-domain :: write
      ( "During opening of the switches for testing the following symptoms occurred:" ) ]
    [ FOR ALL symptom in slot1 of diagnosis
      < [ power-domain :: write
        ( "  a on a" switch of symptom fault of symptom ) ] > ]
    [ power-domain :: write
      ( "This is not a situation that is diagnosable in the existing rule set." ) ]
    [ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ]
    [ FOR ALL symptom in slot1 of diagnosis
      < [ power-domain :: out-of-service ( switch of symptom ) ] > ] >
;

00

```

```

00 MF-diag-13 (unexpected-too-many-symptoms-flip-top)
00   Diagnosed in MF-rule6.4
00
00   MF-diag-13
00   FOR ALL diagnosis in diagnosis-set
00     WHERE [ name of diagnosis = unexpected-too-many-symptoms-flip-top ]
00     < ::>
00       [ diagnosis-set = diagnosis-set MINUS diagnosis ]
00       [ power-domain :: write ( "~a tripped on ~a."
00         switch of top-symp of diagnosis
00         fault of top-symp of diagnosis ) ]
00       [ power-domain :: write
00         ( "During flipping of the top switch for testing the following symptoms occurred:" ) ]
00       [ FOR ALL symptom in slot1 of diagnosis
00         < [ power-domain :: write
00           ( " ~a on ~a switch of symptom fault of symptom ) ] > ]
00       [ power-domain :: write
00         ( "This is not a situation that is diagnosable in the existing rule set." ) ]
00       [ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ]
00       [ FOR ALL symptom in slot1 of diagnosis
00         < [ power-domain :: out-of-service ( switch of symptom ) ] > ] >
00
00 ;
00
00 MF-diag-14 (retrip-on-flip)
00   Diagnosed in MF-rule6.5
00
00   MF-diag-14
00   FOR ALL diagnosis in diagnosis-set
00     WHERE [ name of diagnosis = retrip-on-flip ]
00     < [ fault of top-symp of diagnosis = over-current ]
00     ::>
00       [ diagnosis-set = diagnosis-set MINUS diagnosis ]
00       [ power-domain :: write ( "~a tripped on ~a."
00         switch of top-symp of diagnosis
00         fault of top-symp of diagnosis ) ]
00       [ power-domain :: write ( "During testing ~a retripped on ~a."
00         switch of top-symp of diagnosis
00         fault of top-symp of diagnosis ) ]
00       00 high-impedance or low-impedance (over-current or fast-trip)
00       [ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
00       [ power-domain :: write ( " Most Likely:" ) ]
00       [ power-domain :: write
00         ( " High impedance short in cable below switch, switch output of switch, or the" ) ]
00       [ power-domain :: write ( "switch input of one of the lower switches." ) ]
00       [ power-domain :: write ( " Less Likely:" ) ]

```

```

[ power-domain :: write ( " Current sensor in switch reading high." ) ]
[ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ] >
;
00
00 MF-diag-15 (retrip-on-flip)
00 Diagnosed in MF-rule6.5
00
00 Need to look at the possibility of some bottom level switches below
00 the top switch that may have tripped on fast trip due to energy storage.
00
MF-diag-15
FOR ALL diagnosis in diagnosis-set
WHERE [ name of diagnosis = retrip-on-flip ]
< [ fault of top-symp of diagnosis = fast-trip ]
::>
[ diagnosis-set = diagnosis-set MINUS diagnosis ]
[ power-domain :: write ( "-a tripped on "a."
switch of top-symp of diagnosis
fault of top-symp of diagnosis ) ]
[ power-domain :: write ( "During testing "a retripped on "a."
switch of top-symp of diagnosis
fault of top-symp of diagnosis ) ]
00 high-impedance or low-impedance (over-current or fast-trip)
[ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
[ power-domain :: write ( " Most Likely:" ) ]
[ power-domain :: write
( " Low impedance short in cable below switch, switch output of switch, or the" ) ]
[ power-domain :: write ( "switch input of one of the lower switches." ) ]
[ power-domain :: write ( " Less Likely:" ) ]
[ power-domain :: write ( " Current sensor in switch reading high." ) ]
[ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ] >
;
00
00 MF-diag-16 (unexpected-retrip-during-flip)
00 Diagnosed in MF-rule6.6
00
MF-diag-16
FOR ALL diagnosis in diagnosis-set
WHERE [ name of diagnosis = unexpected-retrip-during-flip ]
< ::>
[ diagnosis-set = diagnosis-set MINUS diagnosis ]
[ power-domain :: write ( "-a tripped on "a."
switch of top-symp of diagnosis
fault of top-symp of diagnosis ) ]
[ power-domain :: write
( "During flipping of the top switch for testing the following symptom occurred:" ) ]
[ FOR ALL symptom in slot1 of diagnosis

```

```

    < [ power-domain :: write
      ( " "a on "a" switch of symptom fault of symptom ) ] > ]
  [ power-domain :: write
    ( "This is not a situation that is diagnosable in the existing rule set." ) ]
  [ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ]
  [ FOR ALL symptom in slot1 of diagnosis
    < [ power-domain :: out-of-service ( switch of symptom ) ] > ] >
;
00
00 MF-diag-17 (not-found-no-levels)
00   Diagnosed in MF-rule6.7
00
MF-diag-17
FOR ALL diagnosis in diagnosis-set
WHERE [ name of diagnosis = not-found-no-levels ]
< ::>
  [ diagnosis-set = diagnosis-set MINUS diagnosis ]
  [ power-domain :: write ( "'a tripped on "a."
    switch of top-symp of diagnosis
    fault of top-symp of diagnosis ) ]
  [ power-domain :: write
    ( "The switch did not retrip during testing and there are no switches below" ) ]
  [ power-domain :: write ( "this switch." ) ]
  [ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
  [ power-domain :: write ( " Most Likely:" ) ]
  [ power-domain :: write ( " A temporary short that was burned clear." ) ]
  [ power-domain :: write ( " A transient in the load below the switch." ) ]
  [ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ] >
;
00
00 MF-diag-18 (unexpected-trips-during-close-top)
00   Diagnosed in MF-rule6.8.1
00
MF-diag-18
FOR ALL diagnosis in diagnosis-set
WHERE [ name of diagnosis = unexpected-trips-during-close-top ]
< ::>
  [ diagnosis-set = diagnosis-set MINUS diagnosis ]
  [ power-domain :: write ( "'a tripped on "a."
    switch of top-symp of diagnosis
    fault of top-symp of diagnosis ) ]
  [ power-domain :: write
    ( "During the close of the top switch for subsequent testing the following" ) ]
  [ power-domain :: write ( "symtom occurred:" ) ]
  [ FOR ALL symptom in slot1 of diagnosis
    < [ power-domain :: write
      ( " "a on "a" switch of symptom fault of symptom ) ] > ]
  [ power-domain :: write

```

```

( "This is not a situation that is diagnosable in the existing rule set." ) ]
[ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ]
[ FOR ALL symptom in slot1 of diagnosis
  < [ power-domain :: out-of-service ( switch of symptom ) ] > ] >
;
00
00 MF-diag-19 (not-found-cant-test-further)
00   Diagnosed in MF-rule6.11 and MF-rule6.19
00
MF-diag-19
FOR ALL diagnosis in diagnosis-set
WHERE [ name of diagnosis = not-found-cant-test-further ]
< ::>
[ diagnosis-set = diagnosis-set MINUS diagnosis ]
[ power-domain :: write ( "'a tripped on 'a."
                        switch of top-symp of diagnosis
                        fault of top-symp of diagnosis ) ]
[ power-domain :: write
  ( "The fault has not been repeated (and therefore not found )." ) ]
[ power-domain :: write ( "The following switches cannot be tested:" ) ]
[ FOR ALL switch in slot1 of diagnosis
  < [ power-domain :: write ( " 'a" switch ) ] > ]
[ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
[ power-domain :: write ( " Most Likely:" ) ]
[ power-domain :: write ( " A transient short somewhere below 'a."
                        switch of top-symp of diagnosis ) ]
[ power-domain :: write
  ( " A short below one of the switches that were not testable." ) ]
[ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ] >
;
00
00 MF-diag-20 (unexpected-to-many-tops-after-flips)
00   Diagnosed in MF-rule6.13
00
MF-diag-20
FOR ALL diagnosis in diagnosis-set
WHERE [ name of diagnosis = unexpected-to-many-tops-after-flips ]
< ::>
[ diagnosis-set = diagnosis-set MINUS diagnosis ]
[ power-domain :: write ( "'a tripped on 'a."
                        switch of top-symp of diagnosis
                        fault of top-symp of diagnosis ) ]
[ power-domain :: write
  ( "During testing of the switches below 'a, the following symptoms occurred:"
    switch of top-symp of diagnosis ) ]
[ FOR ALL symptom in slot1 of diagnosis
  < [ power-domain :: write
    ( " 'a on 'a" switch of symptom fault of symptom ) ] > ]

```

```

[ power-domain :: write
  ( "This is not a situation that is diagnosable in the existing rule set." ) ]
[ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ]
[ FOR ALL symptom in slot1 of diagnosis
  < [ power-domain :: out-of-service ( switch of symptom ) ] > ] >
;

00
00 MF-diag-21 (found-below)
00   Diagnosed in MF-rule6.14
00
MF-diag-21
FOR ALL diagnosis in diagnosis-set
WHERE [ name of diagnosis = found-below ]
< ::>
[ diagnosis-set = diagnosis-set MINUS diagnosis ]
[ power-domain :: write ( "a tripped on a."
  switch of top-symp of diagnosis
  fault of top-symp of diagnosis ) ]
[ power-domain :: write
  ( "During testing of the lower switches the fault was repeated when a was flipped."
  slot1 of diagnosis ) ]
[ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
[ power-domain :: write ( " Most Likely:" ) ]
[ power-domain :: write
  ( " A short below a and (if the switches between a and a are of the same"
  slot1 of diagnosis slot1 of diagnosis switch of top-symp of diagnosis ) ]
[ power-domain :: write
  ( "rating) a race to determine which switch actually trips." ) ]
[ power-domain :: write
  ( " Otherwise, the switches between a and a have failed current sensors"
  slot1 of diagnosis switch of top-symp of diagnosis ) ]
[ power-domain :: write
  ( "in addition to the short below a." slot1 of diagnosis ) ]
[ power-domain :: out-of-service ( slot1 of diagnosis ) ] >
;

00
00 MF-diag-22 (unexpected-different-top-after-flips)
00   Diagnosed in MF-rule6.15
00
MF-diag-22
FOR ALL diagnosis in diagnosis-set
WHERE [ name of diagnosis = unexpected-different-top-after-flips ]
< ::>
[ diagnosis-set = diagnosis-set MINUS diagnosis ]
[ power-domain :: write ( "a tripped on a."
  switch of top-symp of diagnosis
  fault of top-symp of diagnosis ) ]

```

```

[ power-domain :: write
  ( "During testing of the lower switches the following symptoms occurred:" ) ]
[ FOR ALL symptom in slot1 of diagnosis
  < [ power-domain :: write
    ( " "a on "a" switch of symptom fault of symptom ) ] > ]
[ power-domain :: write
  ( "This is not a situation that is diagnosable in the existing rule set." ) ]
[ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ]
[ FOR ALL symptom in slot1 of diagnosis
  < [ power-domain :: out-of-service ( switch of symptom ) ] > ] >
;
00
00 MF-diag-23 (not-found-all-tested)
00   Diagnosed in MF-rule6.20
00
MF-diag-23
FOR ALL diagnosis in diagnosis-set
WHERE [ name of diagnosis = not-found-all-tested ]
< ::>
[ diagnosis-set = diagnosis-set MINUS diagnosis ]
[ power-domain :: write ( "'a tripped on "a."
  switch of top-symp of diagnosis
  fault of top-symp of diagnosis ) ]
[ power-domain :: write
  ( "The fault has not been repeated (and therefore not found)." ) ]
[ power-domain :: write ( "All the testing that is possible has been done." ) ]
[ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
[ power-domain :: write ( " Most Likely:" ) ]
[ power-domain :: write ( " A transient short somewhere below "a."
  switch of top-symp of diagnosis ) ]
[ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ] >
;
00
00 MF-diag-24 (possible-found)
00   Diagnosed in MF-rule6.23
00
MF-diag-24
FOR ALL diagnosis in diagnosis-set
WHERE [ name of diagnosis = possible-found ]
< [ fault of top-symp of diagnosis = over-current ]
::>
[ diagnosis-set = diagnosis-set MINUS diagnosis ]
[ power-domain :: write ( "'a tripped on "a."
  switch of top-symp of diagnosis
  fault of top-symp of diagnosis ) ]
[ power-domain :: write
  ( "During testing of the lower switches the fault was repeated when "a was closed."
  slot1 of diagnosis ) ]

```



```

[ power-domain :: write ( "POSSIBLE CAUSES:" ) ]
[ power-domain :: write ( " Most Likely:" ) ]
[ power-domain :: write
( " This may be indicative of a an over-current fault being generated by too" ) ]
[ power-domain :: write ( "many loads all in over-current in tandem." ) ]
[ power-domain :: write
( "Since this situation is not very likely, "a is still being declared"
switch of top-symp of diagnosis ) ]
[ power-domain :: write ( "out of service." ) ]
[ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ] >
;
00
00 MF-diag-25 (possible-found)
00 Diagnosed in MF-rule6.23
00
MF-diag-25
FOR ALL diagnosis in diagnosis-set
WHERE [ name of diagnosis = possible-found ]
< [ fault of top-symp of diagnosis = fast-trip ]
::>
[ diagnosis-set = diagnosis-set MINUS diagnosis ]
[ power-domain :: write ( "'a tripped on "a."
switch of top-symp of diagnosis
fault of top-symp of diagnosis ) ]
[ power-domain :: write
( "During testing of the lower switches the fault was repeated when "a was closed."
slot1 of diagnosis ) ]
[ power-domain :: write ( "This should not be possible." ) ]
[ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ] >
;
00
00 MF-diag-26 (unexpected-different-trip-during-closes)
00 Diagnosed in MF-rule6.24
00
MF-diag-26
FOR ALL diagnosis in diagnosis-set
WHERE [ name of diagnosis = unexpected-different-trip-during-closes ]
< ::>
[ diagnosis-set = diagnosis-set MINUS diagnosis ]
[ power-domain :: write ( "'a tripped on "a."
switch of top-symp of diagnosis
fault of top-symp of diagnosis ) ]
[ power-domain :: write
( "During testing of the lower switches the following symptoms occurred:" ) ]
[ FOR ALL symptom in slot1 of diagnosis
< [ power-domain :: write
( " "a on "a" switch of symptom fault of symptom ) ] > ]
[ power-domain :: write

```

```

( "This is not a situation that is diagnosable in the existing rule set." ) ]
[ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ]
[ FOR ALL symptom in slot1 of diagnosis
  < [ power-domain :: out-of-service ( switch of symptom ) ] > ] >
;
00
00 MF-diag-27 (unexpected-new-trips-during-closes)
00   Diagnosed in MF-rule6.25
00
MF-diag-27
FOR ALL diagnosis in diagnosis-set
WHERE [ name of diagnosis = unexpected-new-trips-during-closes ]
< ::>
[ diagnosis-set = diagnosis-set MINUS diagnosis ]
[ power-domain :: write ( "a tripped on a."
                           switch of top-symp of diagnosis
                           fault of top-symp of diagnosis ) ]
[ power-domain :: write
  ( "During testing of the lower switches the following symptoms occurred:" ) ]
[ FOR ALL symptom in slot1 of diagnosis
  < [ power-domain :: write
    ( "a on a" switch of symptom fault of symptom ) ] > ]
[ power-domain :: write
  ( "This is not a situation that is diagnosable in the existing rule set." ) ]
[ power-domain :: out-of-service ( switch of top-symp of diagnosis ) ]
[ FOR ALL symptom in slot1 of diagnosis
  < [ power-domain :: out-of-service ( switch of symptom ) ] > ] >
:
[ diagnosis = :unknown ]
[ diagnosis-set = empty ]

DONE

```

4.3.3 The Soft Fault Expert System

The soft fault expert system is one fairly simple rule group.

```

00
00 The Soft Fault Rule Group
00

```

RULE-GROUP : soft-fault

```

00 first deal with the cases where there is an upper switch that
00 can detect current
SF-Rule1
FOR ALL node in sf-nodes
  WHERE [ upper-switch of node ]

```

```

    [ current-trippable of upper-switch of node = true ]
< [ power-domain :: loose-<
    ( power-domain :: sum-values ( lower-switches of node )
      latest-performance-current-avg of upper-sensor of node ) ]
[ power-domain :: loose-<
    ( power-domain :: sum-values ( lower-switches of node )
      latest-performance-current-avg of upper-switch of node ) ]
::>
[ sf-result = sf-result PLUS upper-switch of node ]
[ analyzed of node = :analyzed ] >
;
SF-Rule2
FOR ALL node in sf-nodes
  WHERE [ upper-switch of node ]
    [ current-trippable of upper-switch of node = true ]
  < [ power-domain :: loose-=
    ( power-domain :: sum-values ( lower-switches of node )
      latest-performance-current-avg of upper-sensor of node ) ]
  [ power-domain :: loose-<
    ( power-domain :: sum-values ( lower-switches of node )
      latest-performance-current-avg of upper-switch of node ) ]
  ::>
  [ sf-result = sf-result PLUS upper-switch of node ]
  [ analyzed of node = :analyzed ] >
;
SF-Rule3
FOR ALL node in sf-nodes
  WHERE [ upper-switch of node ]
    [ current-trippable of upper-switch of node = true ]
  < [ power-domain :: loose->=
    ( power-domain :: sum-values ( lower-switches of node )
      latest-performance-current-avg of upper-sensor of node ) ]
  OR
  [ power-domain :: loose->=
    ( power-domain :: sum-values ( lower-switches of node )
      latest-performance-current-avg of upper-switch of node ) ]
  ::>
  [ analyzed of node = :analyzed ] >
;
@@ Now do the nodes where there is an upper switch but it is not
@@ current trippable
SF-Rule4
FOR ALL node in sf-nodes
  WHERE [ upper-switch of node ]
    [ current-trippable of upper-switch of node = false ]
  < [ power-domain :: loose-<
    ( power-domain :: sum-values ( lower-switches of node )
      latest-performance-current-avg of upper-sensor of node ) ]

```

```

00 Figure this one out!!!
[ power-domain :: loose-<
  ( power-domain :: sum-values ( lower-switches of node )
    latest-performance-current-avg of upper-sensor of upper-node of
    upper-sensor of node ) ]
::>
[ sf-result = sf-result PLUS upper-switch of node ]
[ analyzed of node = :analyzed ] >
;
SF-Rule5
FOR ALL node in sf-nodes
  WHERE [ upper-switch of node ]
    [ current-trippable of upper-switch of node = false ]
  < [ power-domain :: loose-=
    ( power-domain :: sum-values ( lower-switches of node )
      latest-performance-current-avg of upper-sensor of node ) ]
  00 Figure this one out!!!
  [ power-domain :: loose-<
    ( power-domain :: sum-values ( lower-switches of node )
      latest-performance-current-avg of upper-sensor of upper-node of
      upper-sensor of node ) ]
  ::>
  [ sf-result = sf-result PLUS upper-switch of node ]
  [ analyzed of node = :analyzed ] >
;
SF-Rule6
FOR ALL node in sf-nodes
  WHERE [ upper-switch of node ]
    [ current-trippable of upper-switch of node = false ]
  < [ power-domain :: loose->=
    ( power-domain :: sum-values ( lower-switches of node )
      latest-performance-current-avg of upper-sensor of node ) ]
  00 Figure this one out!!!
  [ power-domain :: loose->=
    ( power-domain :: sum-values ( lower-switches of node )
      latest-performance-current-avg of upper-sensor of upper-node of
      upper-sensor of node ) ]
  ::>
  [ analyzed of node = :analyzed ] >
;
00 Finally do the nodes where there is no upper switch
SF-Rule7
FOR ALL node in sf-nodes
  WHERE [ upper-switch of node = empty ]
  < 00 the possible fault has already been checked by rules 4-6
  ::>
  [ analyzed of node = :analyzed ] >
;

```

```
[ FOR ALL node in sf-nodes  
  < [ analyzed of node = :analyzed ] >
```

DONE

4.4 Function Reference

This part documents the functions used by the FRAMES knowledge agent to support algorithmic functions of the expert systems. These are all defined in the `knomad/main/domain-functions.cl` file.

The fault diagnosis expert system uses switches and symptoms in most of its rules. A switch is simply a symbol referring to a switch name. A symptom, however, is a frame with two slots: switch and fault. The following functions make use of both switches and symptoms.

`cluster-symptoms symptom-set &optional not-set` [Function]

`top-symptoms symptom-set &optional not-set` [Function]

`new-diagnosable-symptoms top-symptom which-switch` [Function]

These functions are used to organize symptoms into sets. `cluster-symptoms` clusters the symptoms into related sets as described by the approach taken to multiple faults above. `top-symptoms` returns the list of top symptoms in the given symptom set. These would be symptoms with switches that are not below any other symptoms with switches in the symptom set. *not-set* is used to let the function know if *symptom-set* is a symbol or an actual set of symptoms. If *not-set* is non-nil then the given symptom set is taken to be a set of actual symptoms. Otherwise the symptoms are looked up as a symptoms slot reference off of the given symbol in the database. `top-symptoms` returns the set of top symptoms of the *symptom-set*. `cluster-symptoms` returns a set of symptom sets.

In the case of `new-diagnosable-symptoms` a fault has been found and the possibility of further faults exists. If the switch that needs to be taken out of service (*which-switch*) indicates that there are other subtrees of switches below the level of *which-switch* these other subtrees will be examined for faults as well. `new-diagnosable-symptoms` returns those symptoms that still need to be looked at for further faults.

`get-switches-above switch` [Function]

`get-switches-below switch` [Function]

These functions do exactly what their names imply. `get-switches-above` will return a list of switches that are above the given switch. `get-switches-below` will return a list of switches below the given switch.

open-switches <i>symptom</i>	[Function]
flip-switch <i>switch</i>	[Function]
flip-switches <i>switches</i>	[Function]
close-switch <i>switch</i>	[Function]
close-switches <i>switches</i>	[Function]
reclose-switches <i>top-symptom which-switch</i>	[Function]

These functions are used by FRAMES for fault isolation. **open-switches** takes a symptom and opens all the switches below the switch of the symptom (including the switch). The **flip-switch** and **flip-switches** functions flip the given switches. The **close-switch** and **close-switches** functions close the given switches. **reclose-switches** is used to close the switches between *top-symptom* and *which-switch* in preparation for further possible fault diagnosis.

out-of-service <i>switch</i>	[Function]
send-out-of-services	[Function]
end-contingency	[Function]

out-of-service is used to declare a switch out of service. **send-out-of-services** is then used to actually send these switches to the Symbolics for rescheduling purposes. Finally, **end-contingency** is called to end the current contingency and finish the current diagnosis session.

sum-values <i>switches</i>	[Function]
loose-= <i>arg1 arg2</i>	[Function]
loose-> <i>arg1 arg2</i>	[Function]
loose-< <i>arg1 arg2</i>	[Function]
loose->= <i>arg1 arg2</i>	[Function]

These are simple algorithmic functions to support some of the weakness of the rule system expression capabilities. These functions are all used in the soft fault expert system. **sum-values** is used to add up the amperage average data of the switches for comparison with other values. The loose functions are just like their corresponding =, <, >, and >= functions except that they allow a ten percent margin float between the arguments. This is because the sensors as well as the analog to digital conversion produces error in the actual values and this ten percent factor is used to compensate.

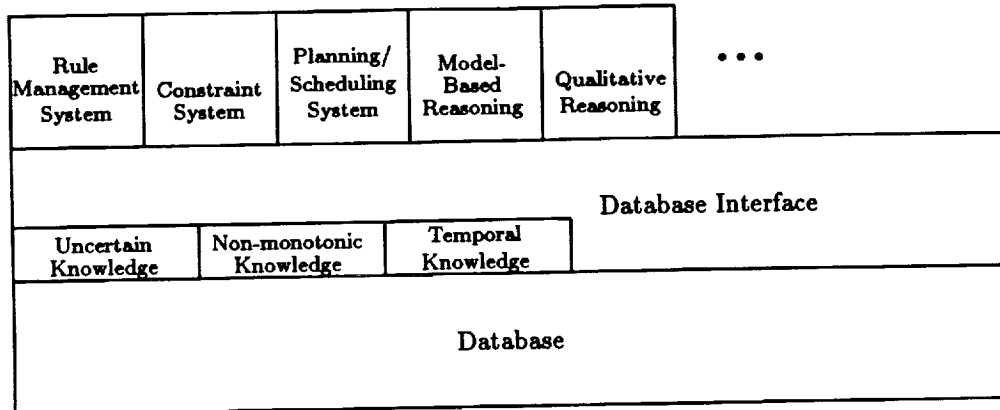


Figure 6: KNOMAD Layered Architecture

5 KNOMAD-SSM/PMAD Technical Reference

KNOMAD-SSM/PMAD, an abbreviation for Knowledge Management and Design applied to the SSM/PMAD domain, is used to represent the domain of SSM/PMAD and provide support for managing multiple knowledge bases.

The architecture of KNOMAD-SSM/PMAD is shown in figure 6. It is a layered architecture that consists of two primary layers. The first layer is the database while the second layer consists of the knowledge base building and inferencing tools. The database layer is further subdivided into two layers. The first of these is the database proper, used for storing tuples. The second of these is the interface to the database. The database interface is where knowledge is structured and abstracted from the primitive tuple representation.

KNOMAD-SSM/PMAD is also data-driven. What this means is that as data is entered into the environment, any knowledge bases that use that data are notified of the change. They can then go off and do what they want to do. By defining the KNOMAD-SSM/PMAD system as both a layered architecture and data-driven, tremendous flexibility has been provided (perhaps at the cost of some efficiency). It is relatively straightforward to add a new knowledge based system tool such as a qualitative reasoning engine. The database proper is also modular. Right now it exists in the same location as KNOMAD-SSM/PMAD. However, there is no reason a distributed database cannot be substituted to provide even more capabilities for distributed artificial intelligence applications.

This section will describe the use of the database and the rule management system. Following that a description of adding tools to KNOMAD-SSM/PMAD will be provided. Further information about KNOMAD-SSM/PMAD may be found in [Riec].

5.1 The Database

The database is used for a dual purpose. It is a database in the normal sense of the word, where data may be stored and accessed. It also serves as working memory for any of the knowledge based systems of the top layer of KNOMAD-SSM/PMAD. For the most part, the user of the database only needs to manipulate that database interface layer of the database. This is in terms of making assertions, matching database values, initializing the database, etc. However, there are some operations that currently must be done at a lower level of the database. The operations of both of these levels will be described here.

The KNOMAD-SSM/PMAD system was developed as a part of supporting the needs of SSM/PMAD. It wasn't developed as a robust, general tool that would work well for all domains. Although in spirit it is a domain independent system, there are problems in the implementation that have not been completely worked out. These are mostly semantic problems that deal with integration between the layers of KNOMAD-SSM/PMAD. We will attempt to point these potential pitfalls out when they are relevant. The pitfalls are only potential since the context of their use will determine how they may affect other parts of the system.

5.1.1 Tuples and Views

The database supports the storage and retrieval of tuples. A tuple is an ordered sequence of (possibly) typed fields. For example, the following are all tuples:

```
(color car-324 white)
(fact fact-23 john)
(car-23)
(eat restaurant-34 joel mud-pie-587 june-23-1990)
(parent (son jane) tarzan)
```

As can be seen, tuples may be arbitrarily complex. Tuples can represent records, as in a relational database, as well as logic.

Tuples are the basic item that is stored and retrieved in the KNOMAD-SSM/PMAD database. The reasoning being that most knowledge based applications are interested in small facts and not large sets of relationships as in typical commercial applications. This does not mean that complicated relational structure cannot be easily represented. Frames, described shortly, are an example of complex relational structure using the tuple representation.

Each field of the tuple may be typed. For example, if the first field is parent, it may be that the final two fields must be people. These types of constraints may be represented using the constraint system of the database.

A further abstraction provided by the database are views. In the traditional database world a view provides a restricted access into the database. For example, if a record of employees with fields of salary, labor grade, age, project, address is in the database a view of the record for normal database users may only allow access to all the fields except for the salary field. Thus the traditional view provides a *window* into the database. In KNOMAD-SSM/PMAD a view is slightly different. A view provides a distinct database for storing data. Thus a view is an organizational entity in KNOMAD-SSM/PMAD.

database:*view* [Variable]

This is the default view that is used by the database for storing and retrieving tuples. It is not a good idea to rebind it, however, users may define their own views (described shortly).

database:view-init *view* [Function]

dnet:dnet-init *view* [Function]

view-init is used to make a new view. *view* should be a symbol that will be bound to the new view. dnet-init is used to initialize a particular view. When the user initializes the database, the user should also use dnet-init to initialize the views created.

5.1.2 Database Constraints

Another aspect of database usage is the ability to add constraints to data in it. Constraints in KNOMAD-SSM/PMAD are used to specify and type fields of tuples.

database:constraint *constraint-pattern* &optional *code* [Function]

constraint is used to specify a constraint on certain tuples in the database. The *constraint-pattern* has the form (in BNF notation):

```
constraint-pattern ::= ( item+ ) | ID
item                ::= constraint-pattern | (:type-decl ID) | (:type-spec ID)
```

The constraint pattern matches a tuple in three ways. One, if the current field of the pattern is ID then the corresponding field of the tuple must be the same as ID. Two, if the current field of the pattern is (:type-spec ID) then the corresponding field must have the type as specified by ID. And three, if the current field of the pattern is (:type-decl ID) then all tuples with the corresponding field must have type ID.

Now that is a bit confusing. The pattern will match all tuples based on the ID and (:type-spec ID) fields. For all those tuples, the fields with (:type-decl ID) are *constrained* to have the correct type.

So, for example, the constraint pattern (`color (:type-spec t) (:type-decl color)`) will match the earlier tuple (`color car-324 white`) and constrain it to have a color in the third field.

Constraints, as specified with this function, will be applied to all tuples as they are added to the database. If the constraint does not pass the tuple, the tuple will not be entered into the database and the storage operation will return `:fail`. When a new constraint is added, all the existing facts that match the constraint and do not pass will be returned by the constraint function. Finally, an optional argument *code* may be provided. If this is provided, the code will be funcalled on tuples as they are added to the database. The code should return a boolean value indicating pass or fail.

Constraints and the higher level tools, particularly the rule management system, have not been will integrated together at this point. It is suggested that constraints not be used at this time in conjunction with the rule management system.

5.1.3 Facts and Frames

The `KNOMAD-SSM/PMAD` database is currently configured to represent facts and frames. Facts are simple tuple of the form (`fact <fact> :value <value>`). This simply says that `<fact>` has value `<value>`.

Frames are very similar to objects in object oriented programming but are more dynamic. Frames allow one to store knowledge in the form of a fixed set of slots (representing the frame) with fillers that can vary from frame to frame. This is like a basic record with fields. However, each slot has structure as well.

Every slot has one or more aspects. There are five aspects defined here: `value`, `mustbe`, `if-needed`, `if-added`, and `constraint`. The `value` aspect represents the value of the slot if it has one. The `if-needed` aspect, if present, provides a LISP procedure for returning a value for the slot (the default procedure is simply to return the `VALUE` aspect) if the slot does not have a `value` aspect. The `if-added` aspect, if present, is a chunk of code executed whenever a slot is written to, that is, whenever a value is added to the slot (or changed). The `constraint` aspect, if present, is a lisp predicate that must be satisfied for a value to constitute a valid filler. The `mustbe` aspect, if present, gives frames or atoms that fillers must either be an instance of, or equal to. The `MUSTBE` aspect can either be a single such item, or a list of `[atom|frame]*`, where only one item in the list must be matched.

The `if-needed`, `if-added`, and `constraint` code are all funcalled with the current value for the slot as the only argument. Other aspects than those described above are legal, but have no special meaning to the frame system. All code is run in an environment where the special variable `self` is bound to the frame itself.

Multiple parents for a frame are defined. The sot aspects for a slot are inherited from the first parent in the parent list that has that slot aspect defined.

An `isnew` method can be defined for frames that executes some LISP code when a frame

is instantiated. The `isnew` code is run in the internal environment for the frame, where `name` is bound to the name of the frame, and `self` is bound to the frame itself. The set of `isnews` collected from all parent frames is run when creating a new frame.

Frames can be defined as follows:

```
(frame :name name
      :parents parent | ( parent* )
      :isnew code
      :slots ({(slot aspect value {aspectvalue}*)}*)
      )
```

```
aspect ::= :if-needed | :constraint | :if-added | :value | :mustbe
```

```
mustbe ::= atom | frame-name | (atom* frame-name*)
```

```
frames:frame &key :name :parents :isnew :slots :view [Macro]
```

```
frames:fcreate-instance parent name [Function]
```

`frame` allows the user to define a new frame with the parameters as discussed earlier. `:view` is used to specify an alternate database view for storing the frame.

`fcreate-instance` is used to instantiate a new frame. This takes two parameters *parent*, and *name*. *parent* must be the name of a previously defined frame. *name* will be the bound to the new frame created and will be the name of the new frame.

```
frames:grind-frame frame [Function]
```

This function can be used to describe a frame, its children, and the slots and their values.

The following LISP session shows an example of defining a frame, creating an instance and asserting a value (more on assertions later).

[10] SSM/PMAD:

```
(frame :name dog :parents (animal) ; animal must be a previously defined frame
      :isnew (format t "~%Creating a dog frame!")
      :slots ((skin :value furry :mustbe organic)
              (nose :value wet :mustbe moisture-level
                    :if-added (lambda (a)
                               (if (eq a 'dry)
                                   (format t "This dog is sick!"))))))
```

```
Creating a dog frame!
<Frame: DOG>
```

[11] SSM/PMAD: (fcreate-instance 'dog 'gandalf)

Creating a dog frame!
<Frame: GANDALF>

[12] SSM/PMAD: (assert! '(frame gandalf nose :value dry))
This dog is sick!
(FRAME GANDALF NOSE :VALUE DRY)

In this implementation making a frame also adds type information to the type hierarchy for use with the constraint feature of the database. All slot and aspect values are stored in the database and may be accessed with the assertion and retrieval functions.

5.1.4 Assertions and Retrievals

Now that the preliminaries have been discussed, tuples, facts and frames, asserting and retrieving tuples from the database can be discussed. There are four assertion and retrieval operations available: `assert!`, `remove!`, `match!` and `retrieve!`. These functions are defined for facts and frames but not for the general tuple.

`dbif:assert! fact &optional who view` [Function]

fact may be a fact or a frame and must be quoted (this is a function). `assert!` uses a primitive database function to actually store the pattern represented by *fact*. If for some reason the pattern cannot be stored because of a failed constraint `:fail` will be returned, otherwise the asserted fact will be returned.

who represents the process performing the assertion is used for purposes of locking mechanisms in the database. If the tuple being updated is locked, only the process represented by the locker may change its value. *view* is used to determine what view to use for the database storage. *view* defaults to `*view*`.

The other primary function of `assert!` is to perform data-driven processing on tuples. Currently this is all hard-coded. When a tuple is asserted, the function checks to see if there are any rule management system dependencies on the tuple. If so `assert!` calls the appropriate function to let the rule management system know about the added tuple. More on this later.

`dbif:remove! fact &optional who view` [Function]

`remove!` is used to remove a fact or frame from the database. If the pattern represented by *fact* is locked then *who* must be the locker of the tuple, otherwise the tuple will not be removed.

remove! is similar to **assert!** in that if there are rule management system dependencies on the tuple being removed, then an appropriate function will be called to let the rule management know about the deleted tuple.

dbif:match! *pattern* *&optional view* [Function]

match! takes a pattern with variables in it and returns a list of tuples in the database that match the pattern. A variable in the pattern is simply a symbol that starts with \$. If there are two variables in the pattern and they are the same, then the value they match must be identical in both places (that is, they must be unifiable).

dbif:retrieve! *pattern* *&optional view* [Function]

retrieve! takes a pattern that may not contain variables in it and returns the tuple in the database that is identical to it, if there is one.

5.1.5 Locks

Locks are used for performing updates to tuples in the database. Locks are very simple to use, one simply locks a tuple, reads and updates the value, and unlocks it.

dbif:lock *pattern* *&optional who view* [Function]

dbif:unlock *pattern* *&optional who view* [Function]

pattern may be any tuple pattern with variables as in the **match!** function described above. This way the user may lock and unlock multiple related tuples. *who* represents the process doing the locking and unlocking.

5.1.6 Initialization

Initialization of the KNOMAD-SSM/PMAD database is done by using the initialization function. Initializing the database will completely initialize it. If the user has any other views they will also have to be reinitialized as described earlier.

dbif:initialize! [Function]

initialize! simply initializes the database.

5.2 The Rule Management System

This subsection describes how to use the rule management system tool of the KNOMAD-SSM/PMAD system. It assumes that the reader has some knowledge of expert systems, and forward and backward chaining. The mechanism for specifying a knowledge base is first presented. This is followed by a description of the rule group. The rule group methods are then discussed, this includes the execution strategy of the rule group, the control strategy and the conflict resolution strategy. Finally the semantics of rules are described. For more technical information about the implementation and perhaps clarification of the discussion here see [Riec] or [Rieb].

5.2.1 The Knowledge Base

Adding a knowledge base to the KNOMAD-SSM/PMAD rule management system must obey fairly rigid syntax. A knowledge base consists of declaring the name of the knowledge base, optionally specifying a domain file that defines the domain, the rule groups of the knowledge base, optional further domain knowledge specific to the knowledge base, which rule groups should begin execution, and finally an end symbol. the exact syntax is defined in the KNOMAD-SSM/PMAD BNF Syntax appendix. The knowledge base generally looks like:

@@ Comments are given by using a '@' symbol. Everything to the end of
@@ line is regarded as comment.

@@ <kb-name> is simply a symbol representing the name of the knowledge base.
KB : <kb-name>

@@ The DOMAIN is optional, but if given, <domain file name> is a filename
@@ that will be loaded to load the domain. The file is loaded as a
@@ LISP file would be loaded using the load function.
DOMAIN : <domain file name>

@@ Rule groups may be directly specified by inserting them directly in the
@@ knowledge base definition. They may also be modularized by placing them
@@ in a separate file in which case the FILE keyword is used.
<rule group>
FILE : <rule group file name>
<rule group>

@@ Domain knowledge is optional and is used to specify further domain
@@ knowledge that is not appropriate in the domain file, but particular
@@ to the knowledge base. Domain knowledge consists of constants, facts,

00 and frames. Constants are specified by specifying symbols separated
00 by semicolons. An example constant is 'true'. Defining a constant
00 asserts a fact into the database that specifies that the constant's
00 value is the same as the constant. The constant 'true' would be
00 asserted as: (fact true :value true).
00 Facts are used to specify values for particular facts. Each fact
00 is separated by a semicolon. An example fact could be:
00 empty = ()
00 This fact would be asserted into the database as: (fact empty :value ()).
00 Finally, frames may also be specified. Frames are specified a little
00 differently than constants and facts. To specify a frame simply put
00 the LISP definitions for frames and their instances here. They are
00 not separated by semicolons but they must be terminated by a period.
00 The frames will be read and evaluated by LISP directly.

Domain-Knowledge :

```
constants : <constants> .  
facts : <facts> .  
frames : <frames> .
```

00 Before the knowledge base is finished the rule groups that should
00 be executed are specified. These are the names of the rule groups
00 as in their specifications. None to any number may be executed
00 in parallel for each knowledge base. If more than one are specified,
00 they must all be on the same line as the begin statement.

Begin : <rule group names>

00 Finally the end symbol is given.

End-KB

One caveat to defining a knowledge base is that all symbols, including punctuation symbols, must be separated by white space. The parser for KNOMAD-SSM/PMAD is a very simple LL(1) (mostly) parser that uses a very primitive lexical analyzer.

Another caveat is that the rule management system operates in the rule-system package of LISP. This means that any functions to be used by the rule groups, symbols in the database, etc., all must be defined to exist (or be visible) in the rule-system package.

rule-system:Pparse-kb *file*

[Function]

This function is used to parse a knowledge base. The *file* must specify the filename where the knowledge base is defined.

rule-system:load-ka ka [Function]

rule-system:kill-ka ka [Function]

load-ka is exactly analogous to parse-kb. It expects a symbol representing the name (pathname) of a knowledge base. It will simply call parse-kb.

kill-ka is used to kill a knowledge agent (which is simply a knowledge base). It is used to stop a knowledge agent from executing and remove any hooks it has in the database.

5.2.2 The Rule Group

The rule group is used to specify a set of rules that are related to one another in some fashion. The rule group consists of six basic parts as specified here:

00 The first part of a rule group is its name.

RULE-GROUP : <rule group name>

00 The rule group rules may be controlled using a transition graph

00 of what rules follow what other rules. This control specification

00 optional. See the FRAMES knowledge base for an example of using this.

CONTROL : <transition table>

00 The control strategy for a rule group may also be specified here.

00 If it is it should be the name of another rule group or a function

00 that will be used as the control strategy for this rule group.

CONTROL-STRATEGY :

00 The conflict resolution strategy is exactly analogous to the control

00 strategy and is optional as well.

CONFLICT-RESOLUTION-STRATEGY :

00 Finally the rules are given. Each rule is separated by a semicolon.

00 The rules will be described shortly.

<rules>

00 After the last rule a colon, optionally followed by a termination

00 condition is specified. The termination condition is a regular

00 rule condition and when it evaluates to true, the rule group is

00 considered to be finished.

: <termination condition>

00 If the rule group is specified in a separate file from the knowledge base

@@ the following must also be in the rule group as the last symbol.
DONE

The semantics of executing a rule group are given in the next section on rule group methods. However, as an introduction the basic execution cycle will be described here for context.

The idea behind a group of rules is to match those rules with data in the database, and from the rules that are matched, select some of them and fire them. This is termed the match-select-fire cycle. The match phase of expert systems is the most expensive. This involves matching the left hand sides (LHS) of rules with data in the database and finding those rules that are *satisfied*. From the resulting set, usually only one, but possibly more, rule(s) are selected. This is the select phase, also called conflict resolution. Finally, those rules that are selected are fired. Firing a rule means that the statements on its right hand side (RHS) are asserted into the database. This cycle continues until the termination condition (basically a LHS of a rule) is satisfied. This description is for a forward chaining expert system. Backward chaining can (almost) be thought of as matching RHSs and asserting LHSs. Basically, backward chaining involves trying to prove a goal, or a RHS. To do this, one *chains* from LHSs to RHSs to find a set of LHSs that are satisfied and by which one could then *chain* from those LHSs to the goal (RHS) of interest.

The rule group definition in KNOMAD-SSM/PMAD provides a lot of flexibility to this model. Specifically, the user may specify how rules can be matched. In a generic expert system, rules match non-deterministically. To get around this most developers put statements into the rules themselves to sequence them properly. The control transition table allows a user to specify up front how rules should be sequenced. The user may also specify different control strategies and conflict resolution strategies for matching and selecting rules.

In the following discussion forward chaining will be assumed for clarity.

Rules are built out of selectors. A selector has the form [referee relation reference]. The *referee* and *reference* are used for specifying database values. For example a referee could be simply *true* or *color of car-34*. *relation* is used to specify how the referee and reference are related to one another. For example, a relation could be *=* or *<=*. When a selector is on the LHS of a rule it is generally used to ask if the referee is related to the reference according to the relation. When a selector is on the RHS of a rule the referee is being given the value of the reference according to the relation (in this case the relation must be something that makes semantic sense, for example *=*).

Selectors may also be quantified. In a quantification, a symbol is bound to the result of an expression and is used in the condition being quantified over. For quantified selectors, if the selector is on the LHS of a rule, the selector is satisfied if the all the bindings are satisfied for the quantified condition in the case of universal quantification, and if there exists one binding which satisfies the quantified condition in the case of existential quantification. If the selector is on the RHS, then either one assertion is done non-deterministically for existential

quantification or, for universal quantification, all assertions are done.

At the next layer are the conditions. A condition is most simply made up of a set of selectors. If on the LHS of a rule, all the selectors must be satisfied. If on the RHS, all the selectors are asserted. Conditions may also be represented by two sets of selectors. The second set either represents a disjunction with the first set or exceptions to the first set. If a disjunction is represented then it is only semantically valid on the LHS of a rule. When a set is used as exceptions then the first set must be satisfied and the *exception* set must not be satisfied for the condition to be satisfied.

Finally rules are made up of a condition on the LHS and RHS of the rule. A rule may be simple or complex. A complex rule is a quantified rule. In quantified rules a variable is bound to the value of an expression but a sub-rule is quantified over instead of a condition.

The syntax of a rule is given in the appendix on the KNOMAD-SSM/PMAD syntax. It is quite complex and perhaps the best way to understand it is in conjunction with the example provided by the FRAMES expert system elsewhere in this document.

5.2.3 Rule Group Methods

The rule group is an object in the database and is specified as follows:

```
(frame :name rule-group
      :slots ((rules :value nil)
              (name)
              (window)
              (quantified-vars)
              (rg-var)
              (plan)
              (plan-state)
              (plan-table :value nil)
              (viable-set :value nil)
              (not-yet-viable-set :value nil)
              (fire-set :value nil)
              (local-variables)
              (rules-fired :value nil)
              (conflict-set :value nil)
              (tickle-set :value nil)
              (satisfied-set :value nil)
              (unsatisfied-set :value nil)
              (cant-fire-set :value nil)
              (fired-set :value nil)
              (untickled-set :value nil)
              (lhs-event))
```

```
(rhs-event)
(*lhs-tickled-queue* :value nil)
(*rhs-tickled-queue* :value nil)
(termination-condition :value nil)
(rules-with-dynamic-lhs-patterns :value nil)
(rules-with-dynamic-rhs-patterns :value nil)
(backtrack-stack :value nil)
(control-strategy :value #'default-control-strategy)
(conflict-resolution-strategy :value
#'default-conflict-resolution-strategy)
(execute :value #'execute1)
))
```

There are many slots to the rule-group definition. Not all of them are currently being used. However, different parts of the execution of the rule group use different slots. The control strategy and conflict resolution strategy are represented as slots on the rule group as well.

When a rule group is executed the following steps are carried out:

1. Initialize the rules of the rule group for execution.
2. Check the termination condition of the rule group. If the termination condition is satisfied the rule group is done.
3. Use the control transition table of the rule group to determine what rules can be matched next. If there is not transition table then all of the rules of the rule group are eligible.
4. Find out which rules of the eligible set are satisfied using the control strategy.
5. Use the conflict resolution strategy to pick a subset of the rules to fire.
6. Fire the selected rules.
7. Loop to number 2.

It is possible to specify a new control strategy or conflict resolution strategy using separate rule groups or user defined functions. However, it is not recommended that the user define a strategy using a rule group for two reasons. One, it will be an order of magnitude slower. Two, the semantics of rules are still being worked on and it may simply not work as expected.

One other caveat of the rule group execution is how the rule management system and the database of KNO MAD-SSM/PMAD are connected. When data is asserted in the database, if the data completes the satisfaction of the LHS of some rule, that rule will be put on the *lhs-tickled-queue* slot of the rule-group frame. This slot is then used by the rest of the execution methods to do forward chaining.

5.2.4 Rule Semantics

The semantics of rules are important for writing rules. As stated above, there are two types of rules, simple and complex. A simple rule has no quantification. A complex rule can have either universal or existential quantification.

A simple rule may be fired once for each execution of a rule group. Once it has been fired it will not be fired again. An existentially quantified rule also will only fire once. However, a universally quantified rule will fire for each set of data that satisfies the rule.

Each time a rule group is executed the rules may be fired in this manner. These semantics are not entirely appropriate for every application. For the FRAMES expert system there is a place where some simple rules need to be fired multiple times. The way this is accomplished is by resetting the fired slot of a rule at certain predictable places in the supporting code for the expert system. The next major version of KNOMAD-SSM/PMAD will have the rule semantics cleaned up so that any application ought to be semantically representable with very little effort.

5.3 Adding Tools to KNOMAD-SSM/PMAD

The rule management system is one of the tools of the top layer of KNOMAD-SSM/PMAD. Adding a tool to the KNOMAD-SSM/PMAD system involves properly integrating it with the KNOMAD-SSM/PMAD database. Recall that the KNOMAD-SSM/PMAD database provides a data-driven environment for the tools. Therefore, adding a tool requires adding hooks into the `assert!` and `retrieve!` functions so that the new tool can be called with the new changes.

The other aspect to consider is what parts of the new tool need to be a part of the database. For example, the rule management system has rule groups as part of the database. Rules are half in the database and half out of the database (defined in a couple of different ways). Generally, the purpose of putting part of the tool into the database is to allow what the tool does to have self-reflexivity to some extent. For the rule system, this allows the specification of control strategies and execution methodologies to be stated as rules by the user.

These two considerations are the only important aspects of adding tools to KNOMAD-SSM/PMAD. It is quite possible, that as this is done extensions to KNOMAD-SSM/PMAD will be defined to help certain aspects of adding tools. For example, since the rule management system is currently the only tool at the top layer of KNOMAD-SSM/PMAD, it is quite possible that the next tool added will want to use rules of the rule management system in some way. The semantics of these types of operations have not yet been defined. It may be that tools can use each other without modifications to KNOMAD-SSM/PMAD. However, it is likely that some modifications will need to be performed.

A Suggested Readings

There are a number of readings that may be useful to better utilize or understand the SSM/PMAD Interface, FRAMES and KNOMAD-SSM/PMAD. These include SSM/PMAD specific references as well as general Artificial Intelligence references.

The first interim final report for this project, [MJA*], as well as volume II are good for understanding the SSM/PMAD in general. These reports talk a lot about how software and hardware functions were implemented as well as specific functionality. [RMA] provides a good description of how the SSM/PMAD implementation provides a number of control loops for operating the power system hardware in an autonomous fashion.

For a general and fairly detailed introduction to Artificial Intelligence [CM] is recommended. [Nil] is also a good reference to AI but is more formal. However, Nilssons's book is a much better book for an understanding of production systems and their semantics.

Two good papers that talk about frames (not the FRAMES expert system) are the seminal paper by Minsky, [Min], and a paper by Hayes, [Hay].

The KNOMAD-SSM/PMAD database is similar to the LINDA model of data and databases but also quite different semantically. Gelernter, [CG], describes the syntax and semantics of the LINDA model as well as how it can be used to solve parallel processing problems in a conceptually elegant manner.

Finally, Charniak, Riesbeck, McDermott, and Meehan provide an accessible reference on AI programming, [CRMM]. There is a lot of good material on data-driven methods, slot and filler databases, production systems, etc.

B KNOMAD-SSM/PMAD BNF Syntax

B.1 Definitions

This appendix describes the syntax of KNOMAD using extended BNF notation. The meanings of the meta-characters are given in the table.

Symbol	Meaning
{	begin optional grouping
}	end optional grouping
	alternative
+	one or more
*	zero or more
ID	analogous to any LISP atom
STRING	a string
NUMBER	a number

B.2 Rule Management System

```
knowledge-base ::= KB : ID { DOMAIN : ID } rule-group+ { domain-knowledge }
                BEGIN : ID+ END-KB

rule-group ::= RULE-GROUP : ID
            { CONTROL : transition-table }
            { CONTROL-STRATEGY : control-strategy }
            { CONFLICT-RESOLUTION-STRATEGY : conflict-resolution-strategy }
            { RG-VAR : ID }
            { QUANTIFIED-VARS : q-vars }
            rules
            { : termination-condition }

domain-knowledge ::= { CONSTANTS : constants . }
                  { FACTS : facts . }
                  { FRAMES : frames . }

control-strategy ::= FUNCTION | ID

conflict-resolution-strategy ::= FUNCTION | ID

termination-condition ::= condition

transition-table ::= ( ( ID ( ID+ ) )+ )

q-vars ::= ID | ID , q-vars

constants ::= ID | ID ; constants

facts ::= fact | fact ; facts

fact ::= ID = ID | ID = ( ID* )

frames ::= frame+

frame ::= ( FRAME :NAME ID
          { :PARENTS parents }
          { :ISNEW FUNCTION }
          { :SLOTS ( ( ID aspect VALUE { aspect VALUE }* )* ) } )

parents ::= ID | ( ID+ )
```

aspect ::= :IF-NEEDED | :IF-ADDED | :CONSTRAINT | :MUSTBE | :VALUE |
 :DISTRIBUTION

rules ::= rule | rule ; rules

rule ::= { condition } ::> rhs |
 { condition } quantifier ID IN expr { WHERE condition } < rule >
 { ELSE condition }

rhs ::= condition { ELSE condition } | { ELSE condition }

quantifier ::= FOR ALL | THERE EXISTS

condition ::= selector+ { choice selector+ }

choice ::= EXCPT | OR

selector ::= [quantifier ID IN expr { WHERE condition } < condition >] |
 [expr { relation reference }]

relation ::= NOT MEMBERIN | NOT UNIQUEIN | MEMBERIN | UNIQUEIN |
 = | <> | >= | <= | > | <

reference ::= expr

expr ::= message | constant | path | expr op expr

constant ::= STRING | NUMBER

op ::= PLUS | MINUS | UNION | TIMES | IDIV | RDIV | MOD

path ::= ID | ID OF path

message ::= path :: ID { (expr+) }

B.3 Frames

frame ::= (FRAME :NAME ID
 { :PARENTS parents }
 { :ISNEW FUNCTION }
 { :SLOTS ((ID aspect VALUE { aspect VALUE }*)*) })

parents ::= ID | (ID+)

aspect ::= :IF-NEEDED | :IF-ADDED | :CONSTRAINT | :MUSTBE | :VALUE |
:DISTRIBUTION

B.4 Database Assertions

In the following, VALUE represents any object.

fact ::= (FACT ID :value VALUE)

frame ::= (FRAME ID slot aspect VALUE)

slot ::= ID

aspect ::= :IF-NEEDED | :IF-ADDED | :CONSTRAINT | :MUSTBE | :VALUE |
:DISTRIBUTION

B.5 Integrity Constraints

i-constraint ::= (item+) | ID

item ::= i-constraint | type-decl | type-spec

type-decl ::= (:type-decl ID)

type-spec ::= (:type-spec ID)

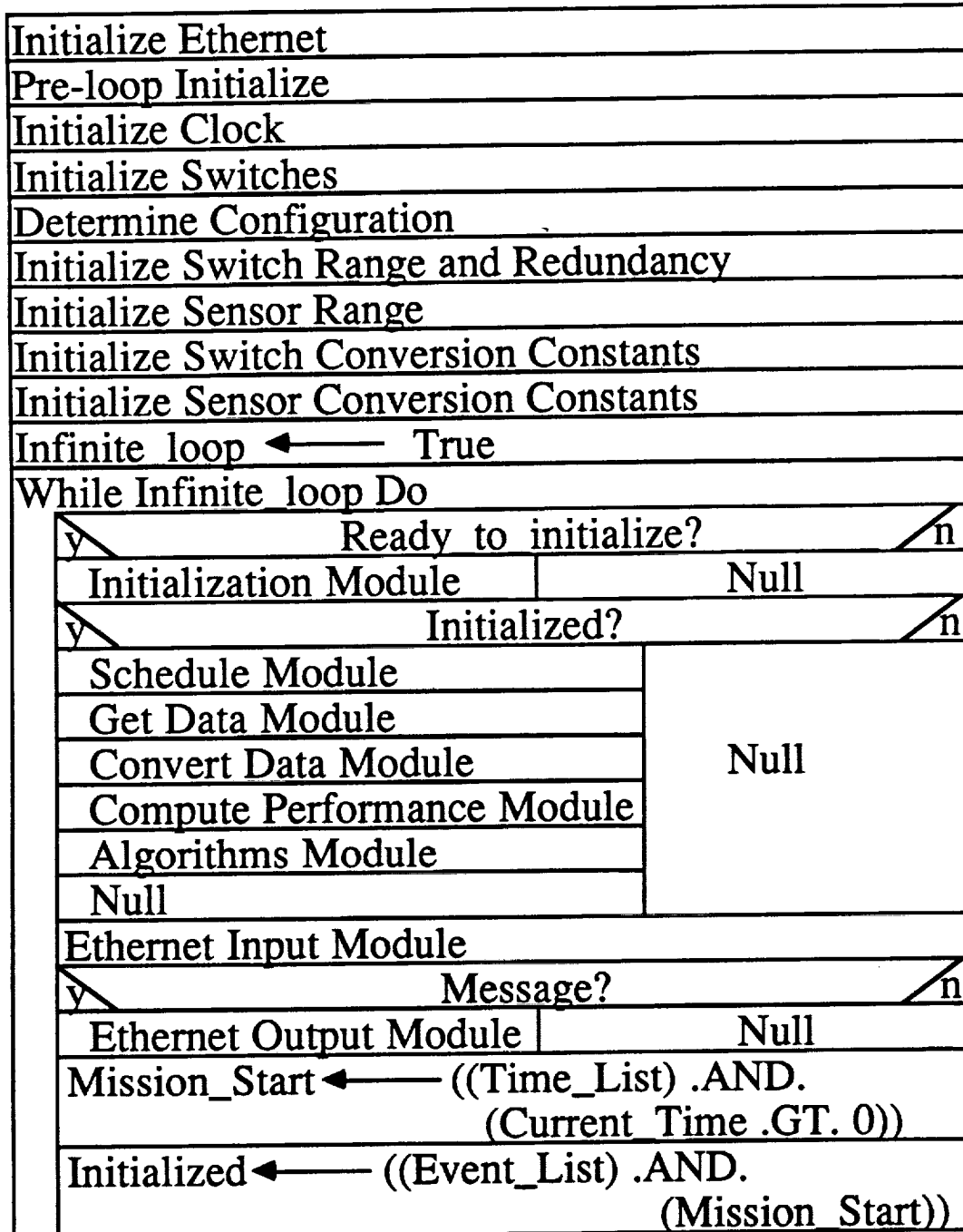
References

- [And] Paul Anderson. *Space Station Common Module Network Topology and Hardware Development*. Contract No. NAS8-36583 Final Report MCR-90-536, Martin Marietta, July 1990.
- [CG] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4), 1989.
- [CM] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence*. Addison-Wesley Publishing Company, 1985.

- [CRMM] Eugene Charniak, Christopher K. Riesbeck, Drew V. McDermott, and James R. Meehan. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, Publishers, second edition, 1987.
- [Hay] P.J. Hayes. The logic of frames. In B.L. Webber and Nils J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 451-458, Morgan Kaufmann, 1981.
- [Min] M. Minsky. A framework for representing knowledge. In P. Winston, editor, *The Psychology of Computer Vision*, pages 211-277, McGraw-Hill, 1975.
- [MJA*] W. Miller, E. Jones, B. Ashworth, J. Riedesel, C. Myers, K. Freeman, D. Steele, R. Palmer, R. Walsh, J. Gohring, D. Pottruff, J. Tietz, and D. Britt. *Space Station Automation of Common Module Power Management and Distribution*. Technical Report Contractor Report 4260, NASA, November 1989.
- [Nil] Nils J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 1980.
- [Riea] Joel D. Riedesel. Diagnosing multiple faults in ssm/pmad. In *Proceedings of the 25th Intersociety Energy Conversion Engineering Conference*, 1990.
- [Rieb] Joel D. Riedesel. *Knowledge Management: An Abstraction of Knowledge Base and Database Management Systems*. Technical Report Contractor Report 4273, NASA, January 1990.
- [Riec] Joel D. Riedesel. Knowledge management: an abstraction of knowledge base and database management systems. In *Proceedings of the Fifth Annual AI Systems in Government Conference*, 1990.
- [RMA] Joel D. Riedesel, Chris Myers, and Barry Ashworth. Intelligent space power automation. In *Proceedings of the Fourth IEEE International Symposium on Intelligent Control*, 1989.



LLP Main Program Revision 3.0



Initialization Module Revision 3.0

Initialize Switches
Pre-loop Initialize
Init list ← True
Initialize Performance Tables
Initialize Switch Range and Redundancy
Initialize Sensor Range
Get Data (initial data set)
Convert Data (initial data set)
Message ← True
Setup Switch Conversion Constant List
Setup A/D Conversion Constant List
Setup Switch/Sensor Configuration List

Schedule Module Revision 3.0

Clear Contingency ← False	
((New_Priority_List) .AND. y (New Priority Time > Current Time))? /n	
Enable New Priority List	Null
((New_Event_List) .AND. y (New Event Time > Current Time))? /n	
Enable New Event List	Null
Set MaxEvent	
Clear Contingency ← True	
While ((Current_Event .LE. MaxEvent) .AND. (Current Event Time .LE. Current Time)) Do	
Process Event	
Increment Current Event	
y Clear Contingency /n	
Contingency ← False	Null

Process Event Revision 3.0

Determine SIC (Based on Switch number)	
Setup Switch Command	
Determine if Switch is Held	
Determine Type of Event	
y\	((Not Held) .OR. (Contingency Op))? /n
Execute Event	Null
Setup Switch Performance list	
y\	Fault Test? /n
Check For Trips	Null

Get Data Module Revision 3.0

y \	SIC A Present?	/n
	Send Switch Data Request to SIC A	Null
	Read SIC A Response	
	Time Stamp SIC A Response	
y \	SIC B Present?	/n
	Send Switch Data Request to SIC B	Null
	Read SIC B Response	
	Time Stamp SIC B Response	
y \	Sensors Available?	/n
	Send Sensor Data Request to Sensor SIC	Null
	Read Sensor SIC Response	
	Time Stamp Sensor SIC Response	

Convert Data Module Revision 3.0

y \ SIC A Present? / n	
Do for all switches on SIC A	
y \ Switch Tripped? / n	Null
Record Trip Data	
Convert Switch Current	
Switch to Redundant	
y \ New Trip? / n	
Setup Switch	Null
Perf. Send	
y \ SIC B Present? / n	
Do for all switches on SIC B	
y \ Switch Tripped? / n	Null
Record Trip Data	
Convert Switch Current	
Switch to Redundant	
y \ New Trip? / n	
Setup Switch	Null
Perf. Send	
y \ Sensors Available? / n	
Do for all Sensors	Null
Convert Sensor Data	
y \ New Trip Info? / n	
Setup Switch Message	Null
Setup Sensor Message	
Setup Switch Performance Message	
Message ← True	

Compute Performance Module Revision 3.0

y\	New Sensor Performance Interval?		/n
Start Time ← End Time		Null	
Do for all sensors			
Reset Vdc / Idc / Power Statistics			
Reset Energy Consumed			
Delta Time ← Sensor Time - End Time			
Do for all sensors			
Update Vdc / Idc / Power Statistics			
Update Energy Consumed			
Do for all switches			
y\	New Switch Performance Interval?		/n
Start Time ← End Time		Null	
Reset Current (Amperage) Statistics			
Delta Time ← SIC Time - End Time			
Update Current (Amperage) Statistics			

Algorithms Module Revision 3.0

y\	Sensors Present?		/n	
Do for all Sensors			Null	
Range Check Sensor Readings				
y\	SIC A Exists?		/n	
Do for all Switches on SIC A			Null	
y\	Time to Send Switch Performance Data?			/n
Setup Switch Performance Send		Null		
y\	Current out of Profile Limits?			/n
Setup Fault Message		Null		
y\	SIC B Exists?		/n	
Do for all Switches on SIC B			Null	
y\	Time to Send Switch Performance Data?			/n
Setup Switch Performance Send		Null		
y\	Current out of Profile Limits?			/n
Setup Fault Message		Null		
Soft Fault Module				

Soft Fault Module Revision 3.0

y \ Sensors are Present? / n		Null
y \ LLP controls a Load Center / n		
Compute Kirchoff's Current Law Sum for SIC A (KCLsum)	Compare Sensor 0 with Sensor 1	
((KCLsum) .GT. (Tolerance))?	y \ Comparable? / n	
Soft Fault ← True Null	Soft Fault ← True Null	
Compute KCLsum for SIC B	Do for Num ← 2 to 7	
((KCLsum) .GT. (Tolerance))?	Compare Sensor Num with Switch Num	
Soft Fault ← True Null	y \ Comparable? / n	
Soft Fault ← True Null	Soft Fault ← True Null	
Compute KCLsum	Compute KCLsum	
((KCLsum) .GT. (Tolerance))?	y \ ((KCLsum) .GT. (Tolerance))? / n	
Soft Fault ← True Null	Soft Fault ← True Null	
Null		
y \ Soft Fault? / n		
Message ← True	Null	
Setup Soft Fault Message		

Ethernet Input Module Revision 3.0

Check for broken connection	
y	Broken? n
Re-establish Connection	Null
Transaction ← (Ethernet Buffer .NE. Empty)	
While Transaction Do	
Transaction Type of:	Read Transaction
	● Event list:
	Process Event list
	Contingency ← False
	Event List ← True
	● Priority list:
	Process Priority list
	Prio List ← True
	● Time list:
	Set System Time
	Time List ← True
	● Contingency Event list:
	Process Event list
	Contingency ← True
	● Switch Control list:
	Execute Switch Control list
	● Switch Conversion Constant list:
	Process Switch Conversion Constant list
	● A/D Conversion Constant list:
	Process A/D Conversion Constant list
	● Initialization list:
	Ready to initialize ← True
	● Query list:
	Process Query list
Transaction ← (Ethernet Buffer .NE. Empty)	

Process Event List Revision 3.0

Initial Event List?	
Enable New Event List	Enable Future Event List
Null	New Event List ← True

Process Priority List Revision 3.0

Initial Priority List?	
Enable New Priority List	Enable Future Priority List
Null	New Priority List ← True

Set System Time Revision 3.0

Convert Start of Mission Day
Convert Start of Mission Month
Convert Start of Mission Year
Store Start of Mission Date
Convert Start of Mission Hour
Convert Start of Mission Minute
Convert Start of Mission Second
Store Start of Mission Time
Convert Present Day
Convert Present Month
Convert Present Year
Set Operating System Date
Convert Present Hour
Convert Present Minute
Convert Present Second
Set Operating System Time

Execute Switch Control List Revision 3.0

Done ← False	
While (Not Done) Do	
Process Event	
Increment Event Number	
y\ (Status is Normal and Fault Isolation List)? /n	
Null	Done ← True
Set Switch Number	
y\ Last event in Switch Control List? /n	
Done ← True	Null
Message ← True	
Get Data	
Convert Data	
y\ Initialized? /n	
Compute Performance	Null

Ethernet Output Module Revision 3.0

y\	Connection Broken?	/n
	Re-establish connection	Null
y\	Connection Broken?	n
	Switch Message?	/n
	Send Switch Status	Null
	Sensor Message?	/n
	Send Sensor Status	Null
	Temperature Message?	/n
	Send Temperature Sensor Status	Null
	Switch Performance Message?	/n
Null	Send Switch Performance	Null
	Sensor Performance Message?	/n
	Send Sensor Performance	Null
	Switch Configuration Constants Message?	/n
	Send Switch Configuration Constants	Null
	Sensor Configuration Constants Message?	/n
	Send Sensor Configuration Constants	Null
	Configuration Message?	/n
	Send Configuration	Null







Initialization List -

Direction - FRAMES to LLP

Description - An initialization list is sent to the LLP during initial startup or when the automation system wishes to reinitialize.

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
LLP Designator	2	Alphanumeric	Which LLP - 'A' to 'H'

Time List -

Direction - FRAMES to LLP

Description - Time synchronization message for distributed software system.

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
Month Now	2	Numeric	Calendar/clock month
Day Now	2	Numeric	Calendar/clock day
Year Now	2	Numeric	Calendar/clock year
Hour Now	2	Numeric	Calendar/clock hour
Minute Now	2	Numeric	Calendar/clock minute
Second Now	2	Numeric	Calendar/clock second
SOM Month	2	Numeric	Start of Mission month
SOM Day	2	Numeric	Start of Mission day
SOM Year	2	Numeric	Start of Mission year
SOM Hour	2	Numeric	Start of Mission hour
SOM Minute	2	Numeric	Start of Mission minute
SOM Second	2	Numeric	Start of Mission second

Event List -

Direction - FRAMES to LLP

Description - A list of events from the Load Enable Schedule for operation of the breadboard.

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
Effective Time	6	Numeric	Effective time of the event list
Number of Events	2	Packed79	Number of Events
EVENT	19	GROUP	AN EVENT DESCRIPTOR
Time of Event	6	Numeric	Time event is to be initiated
Component	3	Alphanumeric	Identity of component
Event	1	Alphanumeric	F-off, N-on, C-change
Type of Event	1	Alphanumeric	Always N-Normal
Redundancy	1	Alphanumeric	Y-Redundant, N-Not Redundant
Switch to Redundant	1	Alphanumeric	Y-Permission, N-No Permission
Maximum Current	3	Numeric	(0-999) deciAmps
Minimum Current	3	Numeric	(0-999) deciAmps

Priority List -

Direction - FRAMES to LLP

Description - Relative Switch Priority list for switches in an LLP.

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
Effective Time	6	Numeric	Effective time of the priority list
Number of Components	2	Packed79	Number of components
Component	3	Alphanumeric	Identity of component

Contingency Event List -

Direction - FRAMES to LLP

Description - A new Event List sent in response to a contingency situation.

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
Effective Time	6	Numeric	Effective time of the event list
Number of States/Events	2	Packed79	Number of (States + Events)
 STATE	 19	 GROUP	 A STATE DESCRIPTOR
Time of Event	6	Numeric	Always 000000
Component	3	Alphanumeric	Identity of component
Event	1	Alphanumeric	F-off, N-on, C-change
Type of Event	1	Alphanumeric	N - Normal, M-Manual
Redundancy	1	Alphanumeric	Y-Redundant, N-Not Redundant
Switch to Redundant	1	Alphanumeric	Y-Permission, N-No Permission
Maximum Current	3	Numeric	(0-999) deciAmps
Minimum Current	3	Numeric	(0-999) deciAmps
 EVENT	 19	 GROUP	 AN EVENT DESCRIPTOR
Time of Event	6	Numeric	Time event is to be initiated
Component	3	Alphanumeric	Identity of component
Event	1	Alphanumeric	F-off, N-on, C-change
Type of Event	1	Alphanumeric	Always N-Normal
Redundancy	1	Alphanumeric	Y-Redundant, N-Not Redundant
Switch to Redundant	1	Alphanumeric	Y-Permission, N-No Permission
Maximum Current	3	Numeric	(0-999) deciAmps
Minimum Current	3	Numeric	(0-999) deciAmps

Switch Control List -

Direction - FRAMES to LLP

Description - A switch command list which is executed immediately. This list is used for immediate source reduction load shedding, fault isolation switch manipulation, and manual intervention.

FIELD	LENGTH	FORMAT	DESCRIPTION
Effective Time	6	Numeric	Effective time of the event list
Number of Events	2	Packed79	Number of Events
EVENT	19	GROUP	AN EVENT DESCRIPTOR
Time of Event	6	Numeric	Not Used
Component	3	Alphanumeric	Identity of component
Event	1	Alphanumeric	F-off, N-on, C-change
Type of Event	1	Alphanumeric	F-Fault Isolation H-Hold until Contingency List M-Manual Control R-Release Manual Control
Redundancy	1	Alphanumeric	Y-Redundant, N-Not Redundant
Switch to Redundant	1	Alphanumeric	Y-Permission, N-No Permission
Maximum Current	3	Numeric	(0-999) deciAmps
Minimum Current	3	Numeric	(0-999) deciAmps

Switch Conversion Constants List -

Direction - FRAMES to LLP

Description - A switch conversion constant list allows the user to tweak the conversion constants used for determining amperage through an RPC. (RBIs presently do not have amperage sensors.)

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
Number of Constants	4	Integer	No. of new switch conversion constants
CONSTANT	11	GROUP	CONSTANT DESCRIPTOR
Component	3	Alphanumeric	Identity of switch getting new conversion constants
Slope	4	Integer	Value in μ Amps
Intercept	4	Integer	Value in μ Amps

A/D Sensor Conversion Constants List -

Direction - FRAMES to LLP

Description - An A/D sensor conversion constant list allows the user to tweak the conversion constants used for determining amperage, voltage, and temperature through a given sensor set.

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
Number of Constants	4	Integer	No. of new sensor set conversion constants
CONSTANT	35	GROUP	CONSTANT DESCRIPTOR
Component	3	Alphanumeric	Identity of sensor set getting new conversion constants
I-Slope	4	Integer	Value in mAmps
I-Intercept	4	Integer	Value in mAmps
V-Slope	4	Integer	Value in mVolts
V-Intercept	4	Integer	Value in mVolts
P-Slope	4	Integer	Value in mWatts
P-Intercept	4	Integer	Value in mWatts
T-Slope	4	Integer	Value in mDegrees
T-Intercept	4	Integer	Value in mDegrees

Query List -

Direction - FRAMES to LLP

Description - A query list may be sent to the LLP to ask for switch status, sensor status, temperature sensor status, configuration data, switch conversion constant data, or sensor conversion constant data.

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
Query type	1	Alphanumeric	A-A/D Conversion Constants C-Configuration F-Offgoing Switch Status N-Ongoing Switch Status Q-Quiescent State Request R-Switch Status S-Sensor Status T-Temperature Sensor Status W-Switch Conversion Constants
Component	3	Alphanumeric	Only used for Query type 'F' , 'N'

Switch Status List -

Direction - LLP to FRAMES

Description - A switch status list is sent to FRAMES when a fault occurs and also in response to a query or switch command list. In addition, this list is sent to FRAMES whenever a switch changes position.

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
Switch number	4	Integer	Used in fault isolation only
Anomalous	1	Alphanumeric	Y-Any fault or warning set N-All OK (nothing set)
LLP_Flags	4	Integer	Bit Defined
bit 0		bit	Quiescent
bit 1		bit	Acknowledge
bit 2 - bit 31		bit	Not Used
Number of switches	4	Integer	Number of switches
SWITCH STATUS	18	GROUP	SWITCH STATUS DESCRIPTOR
Switch component	4	Integer	Identity of switch
Switch position	1	Alphanumeric	F-off, N-on, T-trip'd, U-Unavailable
Switch hold type	1	Alphanumeric	N-Normal, M-Manual, H-Contingency
Status word	4	Integer	Bit Defined (True if set)
bit 0		bit	Anomalous flag
bit 1		bit	Surge Current Trip
bit 2		bit	Over Current Trip
bit 3		bit	Under Voltage Trip
bit 4		bit	Ground Fault Trip
bit 5		bit	Over Temperature Trip
bit 6		bit	Fast Trip
bit 7		bit	Already tripped flag
bit 8		bit	Already on flag
bit 9		bit	Already off flag
bit 10		bit	Not Used
bit 11		bit	Not Used

bit 12	bit	SIC not present flag
bit 13	bit	Generic Card not present flag
bit 14	bit	Not enough power available flag
bit 15	bit	Could not schedule flag
bit 16	bit	Switched to Redundant flag
bit 17	bit	Switch has been shed flag
bit 18	bit	Unable to Command flag
bit 19	bit	Current Overrange flag
bit 20	bit	Out of Current limits flag
bit 21	bit	Over Temperature warning flag
bit 22 - bit 31	bit	Not Used
Amperage	4	Integer
		Current through switch (dAmps)
Trip Tag	4	Integer
		0 - Ignore
		Number-Number of Trip for LLP

Sensor Status List -

Direction - LLP to FRAMES

Description - A sensor status list is sent to FRAMES when a fault occurs and also in response to a query list. In addition, this list is sent to FRAMES on a temporal basis.

FIELD	LENGTH	FORMAT	DESCRIPTION
Number of sensor sets	4	Integer	Number of sensor sets
SENSOR SET	20	GROUP	SENSOR SET DESCRIPTOR
Sensor set component	4	Integer	Identity of sensor set
Amperage	4	Integer	Amperage reading (dAmps)
Voltage	4	Integer	Voltage reading (Volts)
Power	4	Integer	Power reading (Watts)
State	4	Integer	Bit defined
bit 0		bit	Amperage out of Range
bit 1		bit	Voltage out of Range
bit 2		bit	Power out of Range
bit 3 - bit 31		bit	Not Used

Temperature Sensor Status List -

Direction - LLP to FRAMES

Description - A temperature sensor status list is sent in response to a query list.

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
Number of temperature sets	4	Integer	Number of temperature sets
TEMPERATURE SET	8	GROUP	TEMPERATURE SET DESCRIPTOR
Temperature set component	4	Integer	Identity of temperature set
Temperature	4	Integer	Temperature reading (degrees)

Switch Performance List -

Direction - LLP to FRAMES

Description - A switch performance list is sent to FRAMES whenever a switch changes position and on a temporal basis.

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
Number of switches	4	Integer	Number of switches
AMPERAGE DATA	28	GROUP	A SWITCH PERFORMANCE DESCRIPTOR
Component	4	Integer	Identity of Switch
Start time	4	Integer	Start of Performance Interval (sec)
End time	4	Integer	End of Performance Interval (sec)
Average current	4	Integer	Time based averaged current (dAmps)
Maximum current	4	Integer	Maximum interval current (dAmps)
Minimum current	4	Integer	Minimum interval current (dAmps)
Maximum time	4	Integer	Time of Max. current reading (sec)
Minimum time	4	Integer	Time of Min. current reading (sec)

Sensor Performance List -

Direction - LLP to FRAMES

Description - A sensor performance list is sent to FRAMES on a temporal basis.

FIELD	LENGTH	FORMAT	DESCRIPTION
Start time	4	Integer	Start of performance interval (sec)
End time	4	Integer	End of performance interval (sec)
Number of sensors	4	Integer	Number of sensors
 SENSOR	 40	 GROUP	 SENSOR PERFORMANCE DESCRIPTOR
Average Voltage	4	Integer	Time based average voltage (Volts)
Maximum Voltage	4	Integer	Maximum interval voltage (Volts)
Minimum Voltage	4	Integer	Minimum interval voltage (Volts)
Average Current	4	Integer	Time based average current (dAmps)
Maximum Current	4	Integer	Maximum interval current (dAmps)
Minimum Current	4	Integer	Minimum interval current (dAmps)
Average Power	4	Integer	Time based average power (Watts)
Maximum Power	4	Integer	Maximum interval power (Watts)
Minimum Power	4	Integer	Minimum interval power (Watts)
Energy Consumed	4	Integer	Energy Consumed (Watt - Hours)

Switch Conversion Values List -

Direction - LLP to FRAMES

Description - A switch conversion constant list allows the user to see the present values of the conversion constants used for determining amperage through an RPC.
(RBIs presently do not have amperage sensors.)

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
Number of Constants	4	Integer	No. of new switch conversion constants
CONSTANT	12	GROUP	CONSTANT DESCRIPTOR
Component	4	Integer	Identity of switch getting new conversion constants
Slope	4	Integer	Value in μ Amps
Intercept	4	Integer	Value in μ Amps

A/D Sensor Conversion Values List -

Direction - LLP to FRAMES

Description - An A/D sensor conversion constant list allows the user to see values of the conversion constants used for determining amperage, voltage, and temperature through a given sensor set.

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
Number of Constants	4	Integer	No. of new sensor set conversion constants
CONSTANT	36	GROUP	CONSTANT DESCRIPTOR
Component	4	Integer	Identity of sensor set getting new conversion constants
I-Slope	4	Integer	Value in mAmps
I-Intercept	4	Integer	Value in mAmps
V-Slope	4	Integer	Value in mVolts
V-Intercept	4	Integer	Value in mVolts
P-Slope	4	Integer	Value in mWatts
P-Intercept	4	Integer	Value in mWatts
T-Slope	4	Integer	Value in mDegrees
T-Intercept	4	Integer	Value in mDegrees

Switch / Sensor Configuration List -

Direction - LLP to FRAMES

Description - This list tells the requestor the configuration of the LLP. This list is sent during initialization and in response to a query list.

FIELD	LENGTH	FORMAT	DESCRIPTION
Sensors Available	1	Alphanumeric	Y-Available, N-Not available
Number of switches	4	Integer	Number of switches
SWITCH	6	GROUP	SWITCH DESCRIPTOR
Switch Number	4	Integer	Identity of switch
Switch type	1	Alphanumeric	1-1 kW RPC, 3-3 kW RPC, R-RBI U-Unavailable S-Unavailable (SIC) *
Switch Position	1	Alphanumeric	F-off, N-on, T-Tripped, U-Unavailable

*Switch type will be set to 'S' only for switch number 0 or 14 if the SIC is unavailable on Bus A or Bus B respectively.

Quiescent Status Message -

Direction - LLP to FRAMES

Description - This message is used to inform FRAMES of a fault in progress at the LLP software level. This message is also sent in response to a quiescent query when the LLP software has reached a quiescent state.

<u>FIELD</u>	<u>LENGTH</u>	<u>FORMAT</u>	<u>DESCRIPTION</u>
Quiescent Status	1	alphanumeric	T - LLP has reached quiescent state. F - LLP has a fault in progress.

Revision E of this Interface Control Document reflects the changeover to the new Intel based LLPs. Since Motorola and Intel store 16 Bit words in memory differently, this Interface Control Document had to be revised to reflect the difference. Anywhere the Switchgear Interface Card returned a 16 Bit word, the ICD has been changed to show the reversal of the high and low bytes. The only other change from revision D is the 1kW/3kW determination for DC RPCs in a standard switchword.

The following are SIC (Switchgear Interface Card) to LLP (Lowest Level Processor) commands, formats, and expected responses. The COMMANDS are messages from the LLP to the SIC. The RESPONSE is the actual data returned from the SIC in response to a command. The LLP will wait for a RESPONSE from the SIC after each command is sent. If no RESPONSE is received within 2 seconds, the SIC card will be considered nonfunctional. All COMMANDS sent to the SIC card will end with a CR (Carriage Return) which flags end of transmission to the firmware on the MVME331 card (intelligent communications controller). All RESPONSES from the SIC will also end with a CR for the same reason. The MVME331 card removes the CR before transmission from the SIC to LLP and from the LLP to the SIC.

NOTES:

The dip switch configuration for SIC is as follows:

- Switch 1 - switch open (off) - bit0 high
- Switch 2 - switch open (off) - bit1 high
- Switch 3 - switch open (off) - bit2 high
- Switch 4 - switch open (off) - bit3 high

The SIC port configuration is as follows:

- Baud rate - 9600
- Data bits - 8
- Stop bits - 1
- Parity - even

Status Format:

```
*****  
*   byte1   *   byte2   *   byte3   *   byte4   *  
*****
```

where: byte1 -> \$30 -- status OK
 -> \$31 -- status NOT OK

byte2 -> cc -- copy of command received
 with MSB bit always set to 1

byte3 -> \$80 -- status OK
 -> \$FF -- unknown command
 -> \$81 -- first byte not a command byte
 -> \$82 -- did not receive first data byte
 -> \$83 -- first data byte msb not high
 -> \$84 -- did not receive second data byte
 -> \$85 -- second data byte msb not high
 -> \$86 -- switch already on
 -> \$87 -- switch already tripped when
 tried to turn it on
 -> \$88 -- switch already off
 -> \$89 -- switch already tripped when
 tried to turn it off
 -> \$8A -- GC Data Valid error when
 getting switch data

NOTE: If the following statuses are received, do not 'download' switch
settings

-> \$8B -- continous buffer overflow
 (reset continous buffer)
-> \$8C -- once buffer overflow
 (redo once buffer)

NOTE: If the following statuses are received, the SIC card must be reset or must use the redundant SIC

- > \$A1 -- SIC character buffer overrun
- > \$A2 -- character overwritten (OE)
- > \$A4 -- parity error from UART (PE)
- > \$A6 -- OE and PE
- > \$A8 -- framing error (FE)
- > \$AA -- FE and OE
- > \$AC -- FE and PE
- > \$AE -- FE and OE and PE
- > \$F7 -- SIC internal memory parity error

byte4-> \$0D -- end of status

Command Word Format:

```
*****  
*   byte1   *   byte2   *   byte3   *   byte4   *  
*****
```

where: byte1 -> cc -- command
byte2 -> dd1 -- first byte of data word
byte3 -> dd2 -- second byte of data word
byte4 -> \$0D -- end of command

Switchword Format:

bit6=0 (switch not tripped)

bit6=1 (tripped)

bit0	current overrange H (1)	tripped overtemp latched H
bit1	S2 solid state switch on H	S2 solid state switch on H
bit2	S1 mech switch on H	S1 mech switch on H
	DC RPC type H (2)	DC RPC type H (2)
bit3	overtemperature H	overtemperature H
bit4	off control input H (3)	off control input H (3)
bit5	on control input H (3)	on control input H (3)
bit7	always 1	always 1
bit8	current (1)	tripped surge current H
bit9	current (1)	tripped fast trip H (4)
bit10	current (1)	spare
bit11	current (1)	spare
bit12	current (1)	tripped overcurrent (i^2t) H
bit13	current (1)	tripped undervoltage H
bit14	current MSB (1)	tripped grnd fault H
bit15	always 1	always 1

(1) RMS current

(2) If switch contains a mech. relay, then mech switch (on H / off L)
If DC RPC (no mech. relay), then DC RPC type (1 kW H / 3 kW L)

(3)	<u>bit5</u>	<u>bit4</u>	<u>RPC command</u>
	0	0	on (error in hardware)
	0	1	on
	1	0	off
	1	1	no change

(4) For DC RPC fast trip not flagged. DC RPC will be in "off" condition, but "commanded on" in fast trip situation.

GC Data Valid word format:

bit0 -> GC Data Valid switch 0 H
bit1 -> GC Data Valid switch 1 H
bit2 -> GC Data Valid switch 2 H
bit3 -> GC Data Valid switch 3 H
bit4 -> GC Data Valid switch 4 H
bit5 -> GC Data Valid switch 5 H
bit6 -> GC Data Valid switch 6 H
bit7 -> always 1
bit8 -> GC Data Valid switch 7 H
bit9 -> GC Data Valid switch 8 H
bit10 -> GC Data Valid switch 9 H
bit11 -> GC Data Valid switch 10 H
bit12 -> GC Data Valid switch 11 H
bit13 -> GC Data Valid switch 12 H
bit14 -> GC Data Valid switch 13 H
bit15 -> always 1

NOTE: L - data valid
H - data not valid

Sensorword Format:

bit0 -> sensor data bit 4
bit1 -> sensor data bit 5
bit2 -> sensor data bit 6
bit3 -> sensor data bit 7
bit4 -> don't care
bit5 -> don't care
bit6 -> don't care
bit7 -> always 1
bit8 -> sensor data bit 0
bit9 -> sensor data bit 1
bit10 -> sensor data bit 2
bit11 -> sensor data bit 3
bit12 -> don't care
bit13 -> don't care
bit14 -> don't care
bit15 -> always 1

A current/voltage sensorword_set consists of 9 sensorwords of the above format for a given current/voltage sensor. The 9 sensorwords will be of the following order:

V rms
I rms
V offset
I offset
V instantaneous
I instantaneous
P instantaneous
P real
frequency

In this document the notation sensorword_set_n will mean the 9 sensorwords of the described sensorword format in the described order for a given voltage/current sensor "n" where n can be sensor/voltage sensor 0 to 15

- 1) COMMAND: command switch off immediately even if
already off or tripped

FORMAT: cc --> \$20

dd1 --> \$80 + j

j -- 7 bits corresponding to the switches as follows:

bit 0 -> switch 0

bit 1 -> switch 1

bit 2 -> switch 2

bit 3 -> switch 3

bit 4 -> switch 4

bit 5 -> switch 5

bit 6 -> switch 6

dd2 --> \$80 + k

k -- 7 bits corresponding to the switches as follows:

bit 0 -> switch 7

bit 1 -> switch 8

bit 2 -> switch 9

bit 3 -> switch 10

bit 4 -> switch 11

bit 5 -> switch 12

bit 6 -> switch 13

RESPONSE:- set up 2 sec timeout

- status as described in the NOTES

2) **COMMAND:** command switch on immediately even if already
on or tripped

FORMAT: cc --> \$21
dd1 --> \$80 + j (j is defined in (1))
dd2 --> \$80 + k (k is defined in (1))

RESPONSE: - set up 2 sec timeout
- status as described in the NOTES

3) COMMAND: reset switch
FORMAT: cc --> \$22
dd1 --> \$80 + j (j is defined in (1))
dd2 --> \$80 + k (k is defined in (1))

RESPONSE: - set up 2 sec timeout
- status as described in the NOTES

4) COMMAND: select GC (all GC select codes will be set to zero)
FORMAT: cc --> \$23
dd1 --> \$86
dd2 --> \$85

RESPONSE: - set up 2 sec timeout
- status as described in the NOTES

- 5) COMMAND: execute SIC firmware reset (does not reset actual set configuration)
- FORMAT: cc --> \$24
dd1 --> \$80
dd2 --> \$80
- RESPONSE: - set up 2 sec timeout
- four bytes of data plus the status as described in the NOTES where the first two bytes give the following data:

bit 0 -> 0 if GC0 connected, 1 if not
bit 1 -> 0 if GC1 connected, 1 if not
bit 2 -> 0 if GC2 connected, 1 if not
bit 3 -> 0 if GC3 connected, 1 if not
bit 4 -> 0 if GC4 connected, 1 if not
bit 5 -> 0 if GC5 connected, 1 if not
bit 6 -> 0 if GC6 connected, 1 if not
bit 7 -> always 1
bit 8 -> 0 if GC7 connected, 1 if not
bit 9 -> 0 if GC8 connected, 1 if not
bit 10 -> 0 if GC9 connected, 1 if not
bit 11 -> 0 if GC10 connected, 1 if not
bit 12 -> 0 if GC11 connected, 1 if not
bit 13 -> 0 if GC12 connected, 1 if not
bit 14 -> 0 if GC13 connected, 1 if not
bit 15 -> always 1

the third byte gives the following data:

- bit 0 -> current SIC switch0 setting
- bit 1 -> current SIC switch1 setting
- bit 2 -> current SIC switch2 setting
- bit 3 -> current SIC switch3 setting
- bit 4 -> 0 if A/D connected, 1 if not
- bit 5 -> don't care
- bit 6 -> don't care
- bit 7 -> always 1

the fourth byte gives the following data:

- bit 0 -> don't care
- bit 1 -> don't care
- bit 2 -> don't care
- bit 3 -> don't care
- bit 4 -> don't care
- bit 5 -> don't care
- bit 6 -> don't care
- bit 7 -> always 1

6) COMMAND: reset continuous buffer

FORMAT: cc --> \$25

dd1 --> \$80

dd2 --> \$80

RESPONSE: - set up 2 sec timeout

- status as described in the NOTES

- 7) **COMMAND:** fill continuous buffer (First use reset continuous buffer then use this command to download code that is to be continuously executed. Code will start executing as soon as the download is started. Up to 80 of these commands may be concatenated before the buffer space is overrun.)

FORMAT: cc --> \$26
 ee1 --> \$80 + q (q is defined as higher 4 bits of
 8-bit code(see sensorword))
 ee2 --> \$80 + r (r is defined as lower 4 bits of
 8-bit code (see sensorword))
 At the end of the command is appended a \$26
 until the last command, then a \$0D is appended.

RESPONSE: - set up 2 sec timeout
 - status as described in the NOTES

- 8) COMMAND: fill once buffer (This command is used to download code that is to be executed only once. Code execution is started by the trigger once buffer command. Up to 80 of these commands may be concatenated before the buffer space is overrun.)

FORMAT: cc --> \$27
ee1 --> \$80 + q (q is defined as (13))
ee2 --> \$80 + r (r is defined in (13))
ee3 --> as defined in (13)

At the end of the of the command or commands is appended a \$0D.

RESPONSE: - set up 2 sec timeout
- status as described in the NOTES

9) COMMAND: get buffered data

FORMAT: cc --> \$29

dd1 --> \$80 + v

(v is defined as:

bit0 -> buffer0

bit1 -> buffer1

bit2 -> buffer2

bit3 -> buffer3

bit4 -> don't care

bit5 -> don't care

bit6 -> don't care)

dd2 --> \$80

RESPONSE: - set up 2 sec timeout

- data of the following format and status as described in
NOTES

HEADER - \$20

\$sssss - three bytes of status

\$8F - dip switch setting for SIC card
(if not \$8F, SIC card not
installed)

\$nnnn - position in loop counter

\$kk - times through loop counter

\$mm - breakpoint

\$22 - start of data

--> 14 switchwords plus 1 GC Data Valid word

NOTE: TM is temperature
multiplexed, TC is
temperature common
(TM is not useful)

temperature sensorwords 0TM,
0TC, 1TM, 1TC, 2TM,
2TC, 3TM, 3TC
frequency sensorword 0
sensor_word_set_0
frequency sensorword 1
sensor_word_set_1
frequency sensorword 2
sensor_word_set_2
frequency sensorword 3
sensor_word_set_3
\$22 - end of buffer
repeat arrowed sections for sensors
4 to 7, 8 to 11, and 12 to 15

--->

10) COMMAND: trigger once buffer

FORMAT: cc --> \$2A

dd1 --> \$80

dd2 --> \$80

RESPONSE: - set up 2 sec timeout

- status as described in the NOTES

11) COMMAND: get power factor and sign (To calculate the power factor use $pf1 = [P_{avg1} / (V_{rms1} * I_{rms1})]$. Use the same calculation to determine $pf2$ using P_{avg2} , V_{rms2} , and I_{rms2} ; if $pf2 < pf1$ denotes capacitive loading; if $pf2 \geq pf1$ denotes inductive loading; ie, voltage leading current)

FORMAT: cc --> \$2B
dd1 --> \$80 + j (j is defined as 0 to \$F depending on sensor pair used)
dd2 --> \$80

RESPONSE: - set up 2 sec timeout
- data defined as six sensor words for the specified in the following order plus status as described in the NOTES.

V rms1
I rms1
P real1
V rms2
I rms2
P real2

12) **COMMAND:** get data for one specified switch a specified number of times

FORMAT: cc --> \$2C

dd1 --> \$80 + j (j is defined as 1 to \$7F depending on the number of times data is specified to be taken -- input buffer must be taken into account)
dd2 --> \$80 + k (k is defined as 0 to \$D depending on the switch specified)

RESPONSE: -set up 2 sec timeout

- data defined as:

j number of 16-bit switchwords plus the status as described in the notes

13) COMMAND: get data for one specified sensor a specified
number of times

FORMAT: cc --> \$2D
dd1 --> \$80 + j (j is defined as 1 to \$EF depending
on the number of times data
is specified to be taked)
dd2 --> \$80 + k (k is defined as 0 to \$F depending
on the sensor specified)

RESPONSE: - set up 2 sec timeout
- data defined as:
j number of sensorword_set_n for the
specified sensor plus the status as described in the
NOTES

14) COMMAND: command switch on checking switch on or tripped status first; if any of the above conditions exist, the switch command for that particular switch or switches is not executed

FORMAT: cc --> \$2E
dd1 --> \$80 + j (j is defined in (1))
dd2 --> \$80 + k (k is defined in (1))

RESPONSE: - set up 2 sec timeout
- status as described in the NOTES

15) COMMAND: command switch off checking switch off
or tripped status first; if any of the above
conditions exist, the switch command for that
particular switch or switches is not executed

FORMAT: cc --> \$2F
dd1 --> \$80 + j (j is defined in (1))
dd2 --> \$80 + k (k is defined in (1))

RESPONSE: - set up 2 sec timeout
- status as described in the NOTES

16) COMMAND: get data for all fourteen switches a specified number of times.

FORMAT: cc --> \$30
dd1 --> \$80 + j (j is defined as 1 to \$7F depending on the number of times data is specified to be taken, input buffer size must be taken into account)
dd2 --> \$80

RESPONSE: - set up 2 sec timeout
- data defined as:
(j times (fourteen switchwords plus GC Data Validword set)) plus the status as described in the NOTES

17) COMMAND: get data for all sixteen sensors one time

FORMAT: cc --> \$31
dd1 --> \$80
dd2 --> \$80

RESPONSE: - set up 2 sec timeout
- data defined as:

sixteen sensorword_set_n plus status
as described in the NOTES

18) COMMAND: get all 16 temperature sensor readings one time

FORMAT: cc --> \$32

dd1 --> \$80

dd2 --> \$80

RESPONSE: - set up 2 sec timeout

- 16 * 2 sensorwords for the temperature sensors
and the status as described in the NOTES

19) COMMAND: get all 16 power factors and signs
(To calculate the power factors see (17))

FORMAT: cc --> \$33
dd1 --> \$80
dd2 --> \$80

RESPONSE: - set up 2 sec timeout
- data defined as 16 * (six sensor words for each sensor
in the following order) plus the status as described in
the NOTES.

V rms1
I rms1
P real1
V rms2
I rms2
P real2

APPENDIX VII RECENTLY PUBLISHED PAPERS RELEVANT TO THE SSM/PMAD TESTBED

CONTENTS

NASA New Technology Disclosure	2
An Architecture For Automated Fault Diagnosis	9
Reactive Autonomous Planning in Spacecraft	16
Autonomous Operation of a Space Station	
Freedom Type Power Testbed	23
A Survey of Fault Diagnosis Technology	32
Intelligent Space Power Automation	48
An Object Oriented Model for Expert System Shell Design	60
Knowledge Management: An Abstraction of	
Knowledge Base and Database Management Systems	73
A Knowledge Base Architecture for Distributed Knowledge Agents	91

NASA NEW TECHNOLOGY DISCLOSURE

1.0 Detailed Description

The automated power system test bed depicted in Figure 1-1 demonstrates a power distribution system where ground crew and astronaut interactions are minimized. Automated portions of the system, leading to its autonomous capability, were developed under NASA/MSFC contract NAS8-36433, "Space Station Automation of Common Module Power Management and Distribution System". Autonomous elements of the system integrate automated intelligent power control software with smart power hardware developed under NASA/MSFC contract NAS8-36583, "Common Module Power System Network Topology and Hardware Development". The system developed on this contract represents state-of-the-art in intelligently controlled autonomous power management and distribution systems. Detailed descriptions of the system can be found in the attached papers.

The purpose of this project was to automate a breadboard level power management and distribution (PMAD) system test bed which possessed many functional characteristics of a specified Space Station power system. The automation system was built upon two versions: first, a 20 kHz 208 volt ac source with redundancy of the power buses; and second, a high voltage (120 to 150 volt) dc source, again with redundancy of the power buses. There are two power distribution control units which furnish power to six load centers which in turn enable power hardware circuits based upon a system generated schedule. This report documents Martin Marietta's progress in developing new technology to build this specified autonomous system. From this, important gains have been achieved in implementing intelligent control and management for these complex power systems.

The Space Station Module Power Management and Distribution (SSM/PMAD) system possesses the capability to perform diagnosis whenever a distribution fault is encountered. The system autonomously reconfigures its operation during run-time and reschedules activities around the fault, rather than just shutting off power to the affected area completely. The key new technology developments which were paramount in producing this operational system were:

- 1) Developing a system architecture which blended deterministic level processing with Artificial Intelligence (AI) processing without producing real-time performance penalties.
- 2) Providing multi-agent interaction through the partitioning of functions into logical groups and sub-groups.
- 3) Combining multi-knowledge agents and deterministic processes into a singular control element within a distributed environment.

1.1 System Architecture

A result of the initial defining work was to separate items as needed into hardware and software elements. The Space Station Module Power Management and Distribution (SSM/PMAD) breadboard hardware consists of two distinct elements: the power control hardware through which current flows to power target loads, and the automation hardware which is made up of computers and process oriented circuit cards.

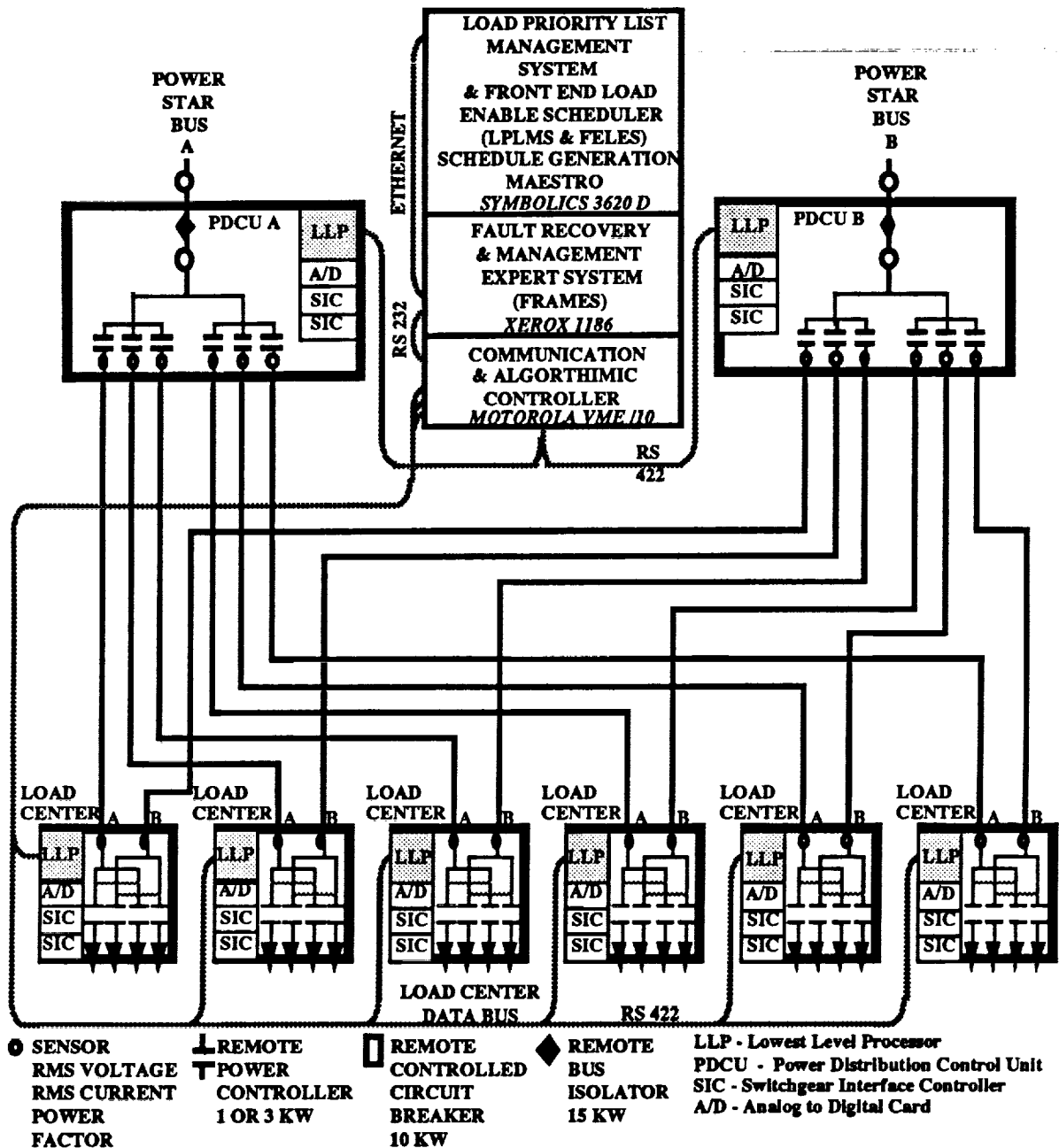


Figure 1-1 SSM/PMAD Test Bed Diagram

Power control hardware consists of analog and digital level hardware units and is considered as part of the power system topology hardware. The automation hardware is part of the automation system and provides the interface between the user, the autonomous functions, the communications and algorithmic controller, the lower level processors (LLPs), and the actual hardware control.

The software architecture takes advantage of the hardware architecture by placing the deterministic elements into the lower level distributed processing elements which interface directly to the power control hardware. Slower operating expert system applications reside farther from the hardware, with the slowest being farthest away in terms of interface layers as shown in Figure 1.3-1. The result is a very crisp command structure close to the actual power control hardware and a more refined reasoning structure where time allows. This provides an environment relatively free of real-time performance penalties.

SSM/PMAD architecture is that of a multi-agent distributed system. The partitioning and distribution of software functions within the architecture provides a real-time capability for interfacing large knowledge processing systems to the time critical power control environment.

1.2 Partitioning of Software Functions

The SSM/PMAD is, by architectural considerations, a multi-agent distributed system. This influenced the partitioning and distribution of the software functions. The user-interface also influenced the location and functional form of the various software elements. The strongest factor influencing the software partitioning was the knowledge content of each functional form.

The Front End Load Enable Scheduler (FELES) provides the user access to the scheduling environment, MAESTRO, and handles returning information from the knowledge based fault management activities. Whenever run-time rescheduling activities are required, FELES initiates scheduling and priority management activities with the appropriate update information.

The Load Priority List Management System (LPLMS) handles initializing and managing priorities of loads. Based upon heuristics, initial priorities for powered loads will change with occurrence of various system events such as changing availability of system power, passage of time, emergencies, and others. These priorities must be managed and allocated in proper ways to assure dependable system performance, and the LPLMS accomplishes this needed function.

MAESTRO is a load scheduling function. It contains basic model knowledge of the overall power system and the required heuristics to ensure correct allocation of resources. The result of MAESTRO's work is the production of a Load Enable Schedule (LES) to be carried out by the LLPs.

The Fault Recovery and Management Expert System (FRAMES) is the backbone of the run-time environment executing the LES. FRAMES diagnoses faults and commands the overall system. FRAMES maintains the system status and provides the autonomous run-time user-interface. FRAMES understands the function and roles of all the operating agents within the SSM/PMAD.

The Communications and Algorithmic Controller (CAC) is the central communication facility for tying the higher level automation hardware and the LLPs together. The various functions which exist on the CAC are bundled to form the Communications and Algorithmic Software (CAS). The primary responsibilities of the CAS are to sort and deliver the LES into its appropriate subcomponent representations for execution by the LLPs, and to stage and deliver data between the FRAMES and the LLPs. It also contains the manual mode operations interface.

Lower Level Functions (LLFs) perform algorithmic management of the LES. They also contain a lower level segment of the FRAMES diagnosis activity which provides rapid limit checking and initial levels of fault condition pattern matching. This provides an innovative implementation of knowledge based functions being executed in a deterministic manner, and then being reasoned on at higher levels. Therefore, the FRAMES/LLF interaction is as that of a worker-manager relationship. The worker performs tasks as defined by the manager, and the manager possesses the capability to reset the workers limits as needed.

Allocation of these software entities to the appropriate hardware and the user's and hardware control access points are shown in Figure 1.2-1.

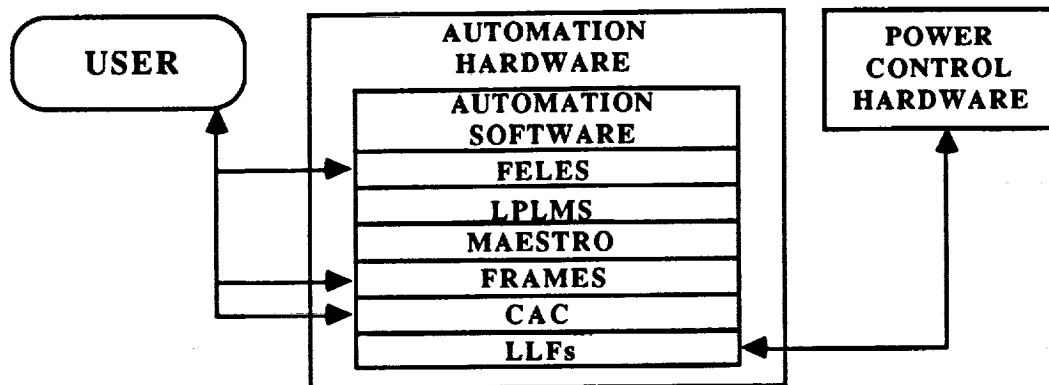


Figure 1.2-1 Automation Software Access and Allocation

1.3 The Singular Control Element

Intelligent control of the SSM/PMAD test bed is provided by the integrated knowledge agents and deterministic functions. In the case of FRAMES, part exists as a knowledge based function, and part exists as a deterministic layer within the LLFs. The control function of the integrated automation system is shown in Figure 1.3-1. Control is parsed into three layers. First, The *Future Parameters* layer, which consists of planning and scheduling functions, supplies either initial or updated activities lists in the form of schedules or tasks. Second, the *Near Real-Time Control* layer which provides fault management and recovery through knowledge based processes, as well as buffering to the distributed environment. Last, the *Real-Time Control* layer, which consists of embedded processing and smart power hardware, provides system level process commanding and data aggregation of the power hardware into knowledge lists.

These three layers provide an overall timing response that fits well into the SSM/PMAD architecture. The control is further strengthened by the proper partitioning of the automation functions. Process times provided by the Real-Time Control in the most critical area occurs in microseconds. Knowledge processing in the Near Real-Time Control may take several seconds, while Future Parameters processing usually ranges up to tens of seconds. Fault protection processing, therefore, occurs at the Real-Time Control layer and

is provided by the LLFs; fault recovery and management are provided by FRAMES at the Near Real-Time Control layer; and planning and scheduling are presently performed by the LPLMS and MAESTRO within the Future Parameters layer. The process complexity and the length of time in process match the layers of needed control and the time in which a

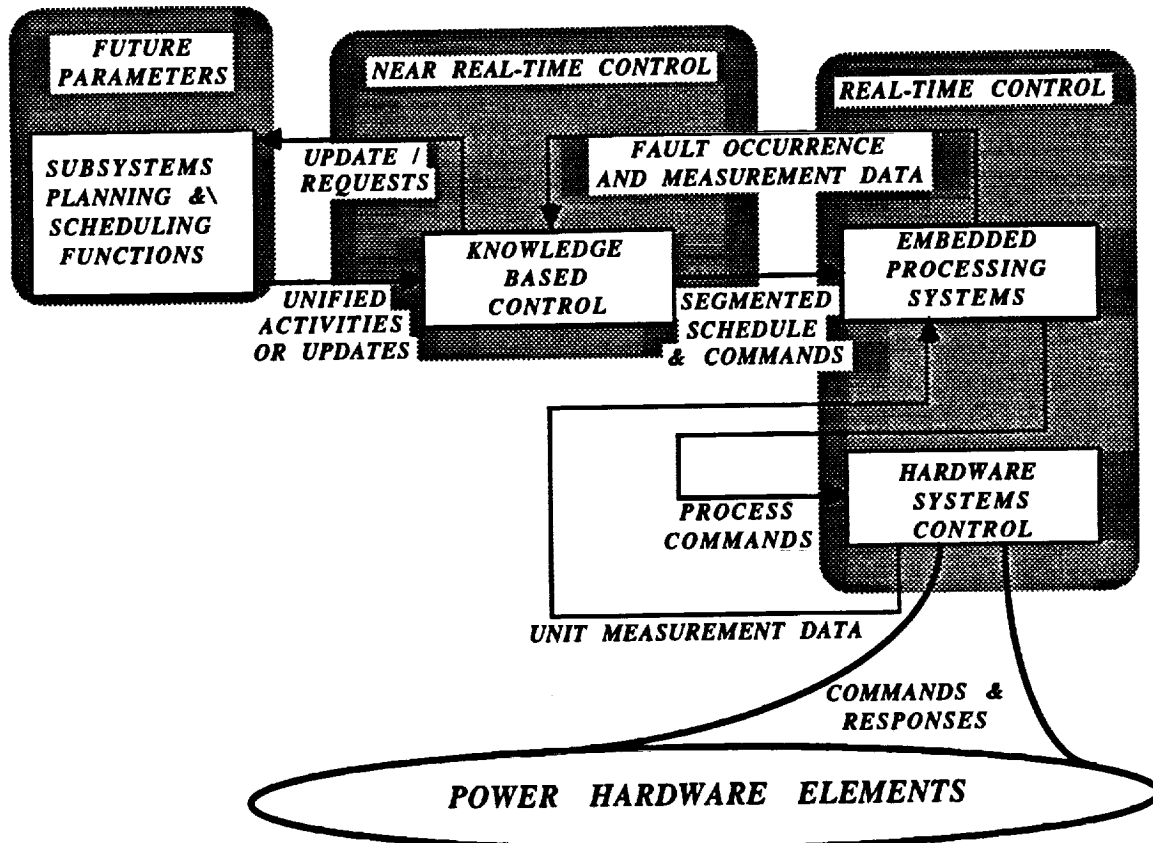


Figure 1.3-1 The SSM/PMAD Control Function

response to a fault or situation is needed. Therefore, the process and its configuration within the overall system appear as one, even though they are separate features.

This provides significant innovation in the way faults may be managed by knowledge based processes introduced into the real-time SSM/PMAD hardware environment. The flow of the execution and management process is quite well defined and is shown in Figure 1.3-2. The initialization part requires inputs and responses from a user. The run-time portion executes autonomously and can reconfigure the system operation in the presence of faults. The fault management process consists of diagnosis and recovery. Diagnosis contains a sub-process providing isolation of the faulted components. Detection of faults occurs at the LLFs and is immediately communicated to higher levels so that diagnosis may begin. Diagnosis may cause opening and closing of switches, as well as gathering of data from what may seem to be unaffected portions of the distributed power control environment. Recovery takes place when diagnosis is complete; however, immediate safing activities usually are provided by smart power hardware or the LLFs whenever a fault is first detected. This information must also be provided whenever diagnosis is underway.

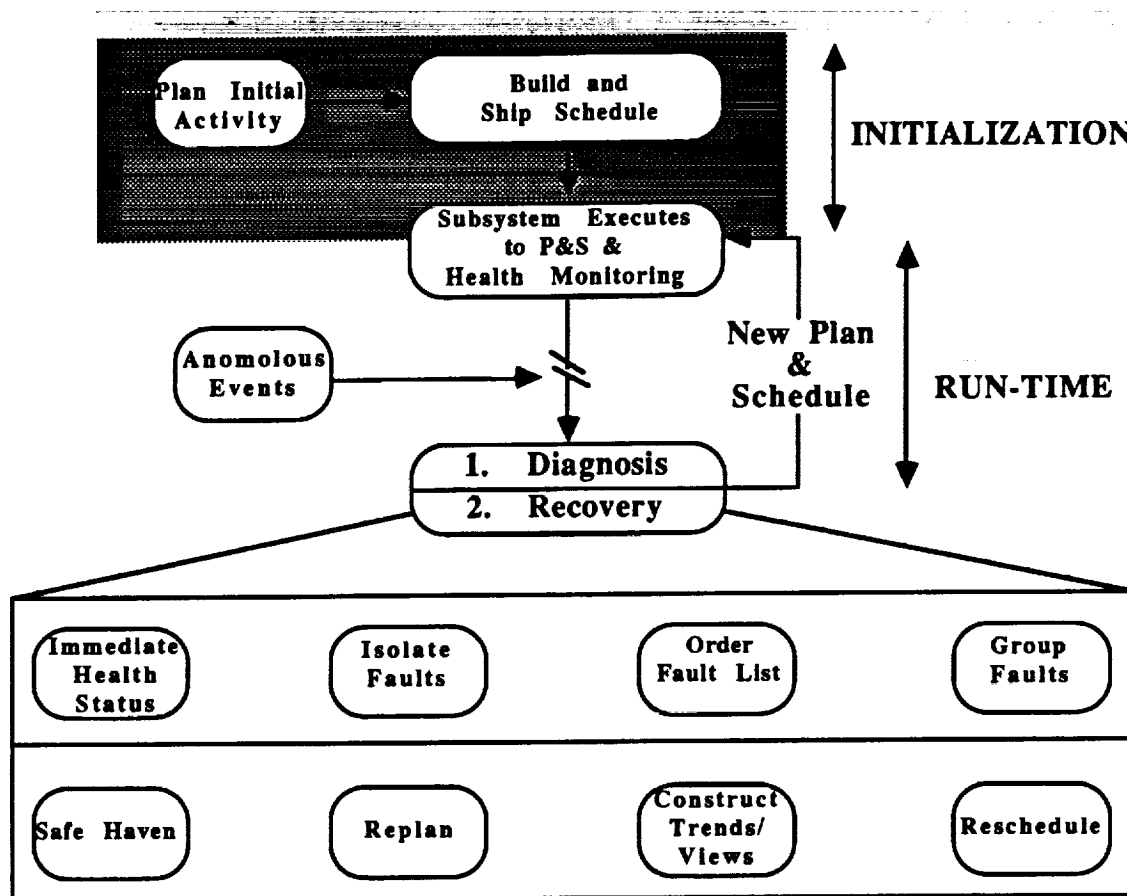


Figure 1.3-2 SSM/PMAD Execution and Management Process Flow

AN ARCHITECTURE FOR AUTOMATED FAULT DIAGNOSIS

A PAPER PRESENTED AT THE IECEC IN AUGUST 1989.

AN ARCHITECTURE FOR AUTOMATED FAULT DIAGNOSIS

Barry R. Ashworth
Martin Marietta Astronautics Group
Denver, Colorado

ABSTRACT

Martin Marietta, under contract to NASA, Marshall Space Flight Center since 1985, is continuing the development of technologies and methodologies for use in the automation of power management and distribution.

Automating the management and control of power distribution systems introduces issues and concerns for the flow of information and control. This is especially true when knowledge-based systems are used in diagnosing faults and trends within the system. Processing functions must obtain the appropriate data from the circuits and hardware which distribute power. Providing the correct mix of hardware control and automated diagnosis with testing introduces system-wide architectural possibilities. In this paper an architecture containing a knowledge-based system which has been successfully used in power system management and fault diagnosis is presented. Architectural issues which effect overall system activities and performance are examined. The knowledge-based system is discussed along with its associated automation implications, and interfaces throughout the system are presented.

1. Nomenclature

AI	Artificial Intelligence
AC, ac	Alternating Current
CAC	Communications and Algorithmic Controller
CAS	Communications and Algorithmic Software
CC(E)	Computation and Control (Engine)
DC, dc	direct current
FELES	Front End Load Enable Scheduler
FRAMES	Fault Recovery and Management Expert System
GC	Generic Controller
Hz, hz	Hertz (cycles per second)
K, k	Kilo (1000)
KW, kw	Kilowatt (1000 watts)
LC	Load Center
LLF	Lowest Level Function
LLP	Lowest Level Processor
LPLMS	Load Priority List Management System
MSFC	(George C.) Marshall Space Flight Center
NASA	National Aeronautics and Space Administration
PDCU	Power Distribution Control Unit
RBI	Remote Bus Isolator
RCCB	Remotely Controlled Circuit Breaker
RPC	Remote Power Controller
SIC	Switch Interface Controller
SSM/PMAD	Space Station Module (Autonomous) Power Management and Distribution

2. Introduction

The commitment to the integration of advanced technologies in the future complex space efforts ([16] and [21]) is mandatory. The advanced architectural considerations [11] encountered in these types of applications presents many opportunities and obstacles which arise when exploring possible solutions to these complex problems. The systems engineering and integration encountered when allocating a broad range of technological disciplines to a subsystem control problem yields unique architectural results within the solution system. Architectures are often overlooked in their cause and content once a system is operational. However, the architecture, along with its cause and content provides the elemental view into system activities and performance.

The Subsystems Application Automation Team within Martin Marietta's Denver Astronautics Group has developed and installed the SSM/PMAD power system automation test bed at the NASA/MSFC Electrical Power Branch in Huntsville, Alabama. A systems engineering approach [4] was used in the development of the test bed from the project start in 1985. The architectural elements of the SSM/PMAD consist of both hardware and software, and their integrated composite, as seen in Figure 1.

This composition gives rise to a function defining methodology known as function partitioning [14] which was used extensively in the SSM/PMAD development. After the functions were partitioned, functional decomposition was used to establish implementable subfunctions which fulfilled the overall architecture. In order to understand the overall system, it is necessary to first understand the system objective and the primary goal within that objective. The objective was to provide autonomous management for an advanced automation utility power management system test bed. The primary goal within the system objective was to be able to manage the power system, autonomously, *even during the occurrence of faults*. This objective and goal taken in the context of a complete power system led to the architecture described here.

Hardware element partitioning gave rise to two fundamental components. First, the Power and Switchgear component [1] which is used to enable power to loads. Second, the Automation hardware component which contains the automation process used to control the switchgear components. The composition of these components is discussed in the respective hardware subsections under Section 3.

In turn, the partitioning process of the software element gave rise to two primary software subcomponents. One, used in the process of direct control very near the switchgear hardware elements, utilized well behaved, quite predictable subfunctions and became the Deterministic component. The other, not so predictable, but well behaved if viewed in the context of knowledge processing functions, formed the Knowledge Based component. These components are further described in their respective subsections under Section 3.

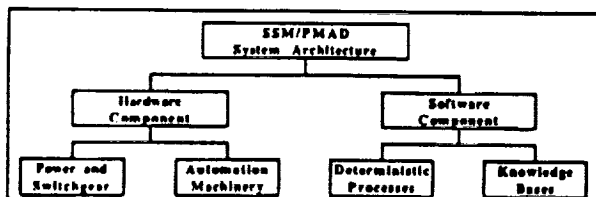


Figure 1 The SSM/PMAD Architectural Hierarchy

3. The SSM/PMAD System Structure

The integrated system structure for the SSM/PMAD contains all hardware and software components. This is shown in Figure 2. The utility load connections reside at the outputs of the LCs. The source power is 208 Volt 20 KHz ac, and the bus topology of the present implementation is a ring¹. When considering the structure of the SSM/PMAD in detail, the various elements must be considered one at a time and then integrated. The elements are the hardware and the software, and to a lesser degree the user interface because the system's objective is autonomous operation. However, even an autonomous system should be rich in user interfaces and should possess a complete manual operation capability. The SSM/PMAD does.

3.1 The Hardware Element

3.1.1 Power and Switchgear Hardware

The hardware element of the SSM/PMAD contains the two fundamental components: power & switchgear hardware and automation hardware. The primary activity of the power & switchgear hardware is to perform instructed switching activities and to provide a current path for enabling power to the utility connections used by varying consumer loads. A detail of the hardware element is shown in Figure 3.

Existing within the power & switchgear hardware component are several subcomponents. The primary subcomponent which is purely power is the power bus itself. Other subcomponents have to do with switchgear. These are the SIC card, the GC card, the A/D card, and the RPC or RCCB. RPCs can either be rated at 3 kw or 1 kw each, depending on their specific application, and the RCCBs are rated at 10 kw. These ratings relate to the amount of power being supplied to different locations in the test bed as can be seen in Figure 2, which also shows the 15 kw RBIs that reside above the PDCU RCCBs. All of this is intended to provide a power distribution architecture which can be implemented and understood within a knowledge base/deterministic software setting which will be described in the next subsection.

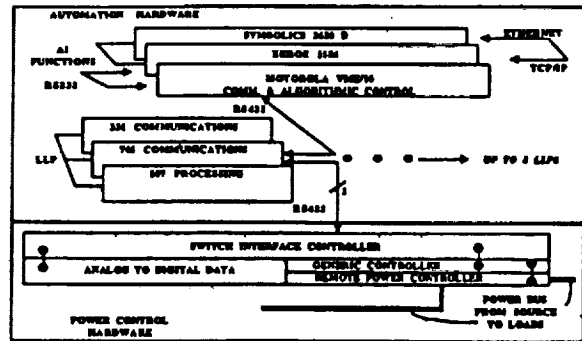


Figure 3 SSM/PMAD Hardware Architectural Element

3.1.2 Automation Hardware

Another component of the hardware element, as can be seen in Figure 3, is the automation hardware. This is composed of the various Computational and Control (CC) engines which provide the physical computing environment for the test bed. There is one physical computing environment for the test bed. It is both distributed and modular, allowing gradual system degradation at the loss of individual LLP CC units.

3.1.2.1 Lowest Level Processors

There are currently up to eight LLPs allowed on the test bed at any one time. However, nothing prevents that number from being much larger. Each LLP is composed of three separate cards: a 107 processing card utilizing a Motorola 68010 micro-processor, a 331 communications card, and a 705 card also used in communications activities. The LLPs communicate with the Motorola VME/10 (CAC) and the lower level SIC cards via an RS422 connection and are the lowest level at which upper level communications requests are routed. Therefore, the LLPs are considered as a minor extension of the upper level processing platforms such as the Motorola VME/10 and the Xerox 1186.

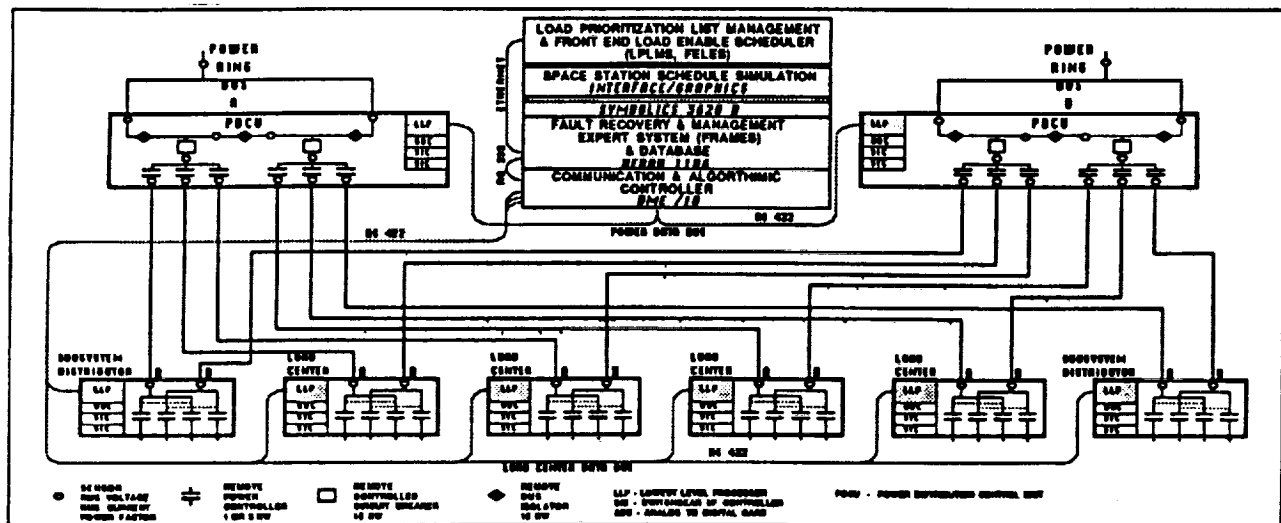


Figure 2 SSM/PMAD Breadboard Structure

¹ On December 14, 1988 a NASA Change Request specifying 120 Volt dc source power was put into effect. Also, since then the bus topology has changed to a STAR.

ORIGINAL PAGE IS
OF POOR QUALITY

3.1.2.2 Communications and Algorithmic Controller

The CAC is used as a communications and routing dispatcher. Communications to upper levels is handled across an RS232 connection. This currently presents no performance degradation due to massive communications activity at the lower levels. However, taken in its entirety, communications surrounding the CAC does present a performance issue which is being examined closely as a place for broad system performance enhancement.

3.1.2.3 LISP Engines

At the upper levels of the automation hardware architecture are the knowledge base (Or AI) environments. These are LISP CCEs which provide great power as development environments for rapid prototyping. One is a Xerox 1186, and the other is a Symbolics 3620D (D for Development). These machines fit in to the development scheme for the SSM/PMAD test bed quite well because they strongly supported the iterative refinement techniques used through vendor offered software tools.

The total hardware architectural element provides a flow. This flow can be understood best in terms of the software architectural element, which will be shown to lay on top of the hardware element, elegantly as an extension.

3.2 The Software Element

Components of the software element are divided logically along the lines of function partitioning and allocation to hardware platforms. These will be described starting once again at the lower levels and working our way up. Figure 2 shows the higher level allocation of software functions such as fault management and scheduling. These software functions are knowledge based components and promulgate the central theme for the autonomous functionality. However, there is also functionality at the CAC and LLPs which should not be overlooked. The SSM/PMAD hardware/software automation breakout is shown in Figure 4.

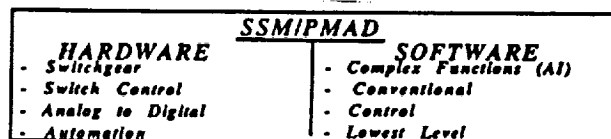


Figure 4 Automation Architecture Breakout

3.2.1 Lower Level Functions

The LLPs reside at the LLPs respectively. These are specialized to deal with the needs of the PDCUs or the LCs. The LLPs perform data acquisition, organize, and format the switch hardware data for use by the upper level knowledge base function at FRAMES. They also are the sole interrogators of switchgear hardware. The LLPs also provide a FRAMES direct extension which performs limit checking on a more rapid basis than could be provided from the upper level FRAMES. This is shown in Figure 5. LLP software development was done using the PASCAL language.

3.2.2 Communications and Algorithmic Software

CAS resides solely on the CAC and provides trafficking control, as well as management of the incoming management software structures for use at the LLPs. It also organizes and buffers data going up to FRAMES from the LLPs. CAS contains elements of the user interface software and provides the complete interface during manual control activities. CAS software was developed using the PASCAL language. CAS activity in the test bed automation software is shown in Figure 5.

3.2.3 Fault Recovery and Management Expert System

FRAMES is the heart of the SSM/PMAD test bed. FRAMES ensures that the load enable schedule which was provided from the scheduling interface is managed properly. If a fault occurs, FRAMES diagnoses the fault ([15], [18], and [20]) and directs system recovery if one is possible. FRAMES has been designed to handle many types of faults which are identified in Table 1. If a fault occurs in the system FRAMES communicates this event over to the Controller which resides as part of the FELES. If FRAMES can recover intact from the fault, this is also communicated. But, if not, the scheduler, MAESTRO, builds a new schedule of events in coordination with the LPLMS which is in turn sent to FRAMES. The new schedule, complete with its associated priority list is then integrated into the test bed by FRAMES, CAS, and each involved LLP. FRAMES is a knowledge based activity and is implemented in the LISP language. The location of the FRAMES knowledge based system is depicted in Figure 5.

3.2.4 Front End Load Enable Scheduler

The FELES interfaces both the user and FRAMES to the scheduling program, MAESTRO. FELES helps to define defaults for activity types and provides the user interface software for activity initialization. FELES is a knowledge based activity and was developed using the LISP programming language. FELES is shown on Figure 5.

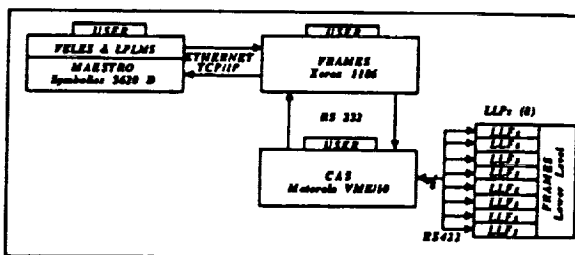


Figure 5 Lower Level Function Extension of Frames

3.2.5 Load Priority List Management System

A list priority structure is important to overall system activity, especially during times of fault occurrence. It is the responsibility of the LPLMS to generate and maintain this priority list for test bed activity. The LPLMS is a knowledge based process and is implemented in LISP. It is shown in Figure 5.

3.2.6 MAESTRO Scheduler

The scheduling is performed by the MAESTRO scheduling component [2]. MAESTRO was developed separately from the SSM/PMAD software. It is a knowledge based activity which is implemented in LISP. MAESTRO's location is shown in Figure 5.

All of the previously described software functions work in unison to provide SSM/PMAD system test bed autonomy. This autonomous function implies a certain integrated control aspect to the overall functionality which has not yet been described. It will be described along with the system operational modes in the following subsections.

3.3 Element Integration, Control, and User Interface

3.3.1 Element Integration

Each of the various hardware and software components for the test bed were integrated to provide an overall system. Figure 5 shows the software to hardware allocation which resides within the system. The communications protocols which were utilized are those shown in Figure 3. The Knowledge based activities were concentrated into units which had unique or shared aspects.

For instance, FRAMES activity is unique and independent of LPLMS and FELES activity once the autonomous operation is initiated. Therefore, it resides on a unique platform, the Xerox 1186. The FELES and LPLMS functions both relate to scheduling and so were placed on the Symbolics hardware platform with MAESTRO. The schedule is allowed to be updated by an event within the autonomous run-time environment or by a user-requested reduction. Schedule updates are affected by priorities which are managed by the LPLMS on a 15 minute basis. The overall system maintains a general knowledge of temporal activity but, in general, all software components are ignorant of event time specifics. A look at the generally integrated architecture of Figure 5 presents a strong argument for not maintaining a synchronous environment.

One issue of element integration which bears mention is that of logic model-based reasoning versus heuristics. There were considerations about primary memory sizes, bus speeds, and communication link speeds which forced the choice of heuristics employed in the FRAMES knowledge base. Fault symptom matrices were developed and employed heuristics derived from knowledge engineering. These allowed the use of heuristics for all cases except the situation of topological structure. FRAMES maintains a simple model for the current system topology.

- 1) Hard Faults - directly measured limit responses.
- 2) Soft Faults - unscheduled use of current.
- 3) Cascaded Faults - faults flowing into faults.
- 4) Incipient Faults - faults about to occur.
- 5) Masked Faults - faults which emerge from faults.

Table 1 Fault Types Handled by FRAMES

The result of the integrated system provides a large processing network. Basically the whole of the system is a network with the exception of the FRAMES component which actually exists partially as a message passing ([9] and [19]) function, loosely coupled among multiple distributed processors. A flow of knowledge based information throughout the system results and is shown in Figure 6.

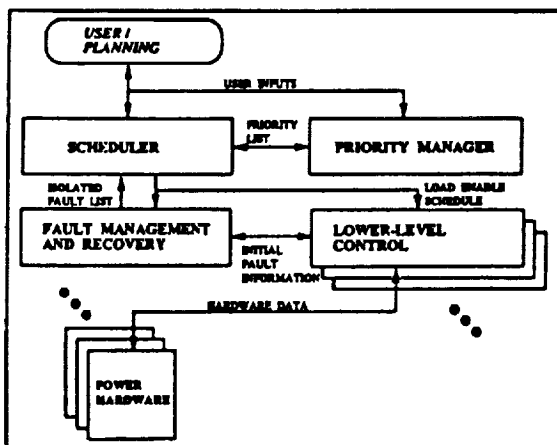


Figure 6 System Knowledge Flow

3.3.2 Control

The SSM/PMAD possesses three operational modes. These are initialization, autonomous activity, and manual activity. The modes and their activities are depicted in Figures 7, 8, and 9. This shows that the user is the highest level of control in the system. However, given the overall system without considering the user's level of control, the test bed structure can be described as a nested control (software control) loop. This is shown in Figure 10. The lowest level of system control exists at the LLPs, and an owning higher level of control exists with FRAMES at the Xerox 1186. This is an important feature which allows the system to be viewed from a limiting situation rather than just a knowledge base interfacing to deterministic level processes.

3.3.3 The User Interface

The user interface consists of the elements shown in Figure 11. One of these elements, the one utilized by FRAMES at the Xerox 1186, displays the system as shown in Figure 12.

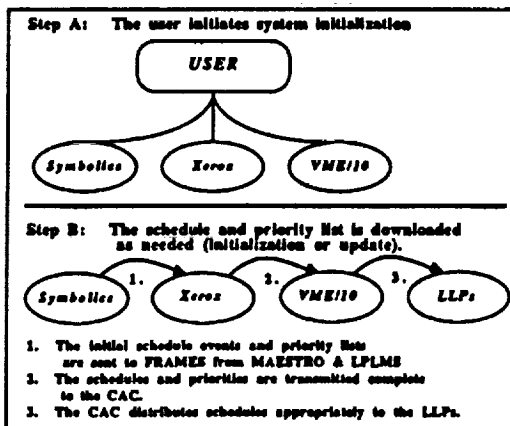


Figure 7 SSM/PMAD System Initialization

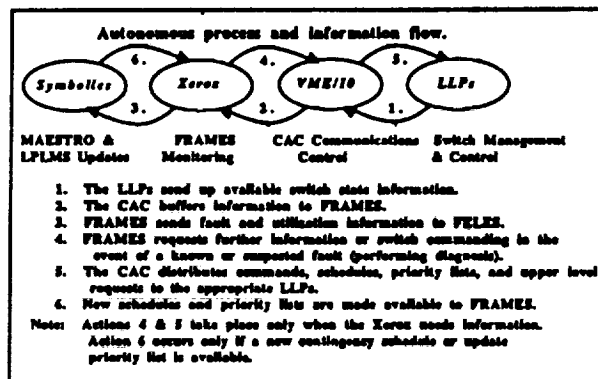


Figure 8 SSM/PMAD Autonomous Mode Operation

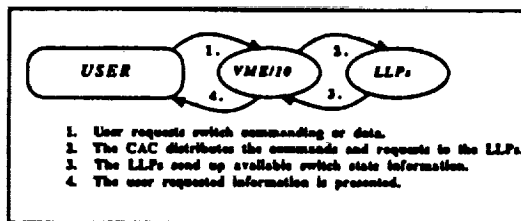


Figure 9 SSM/PMAD Manual Mode Operation

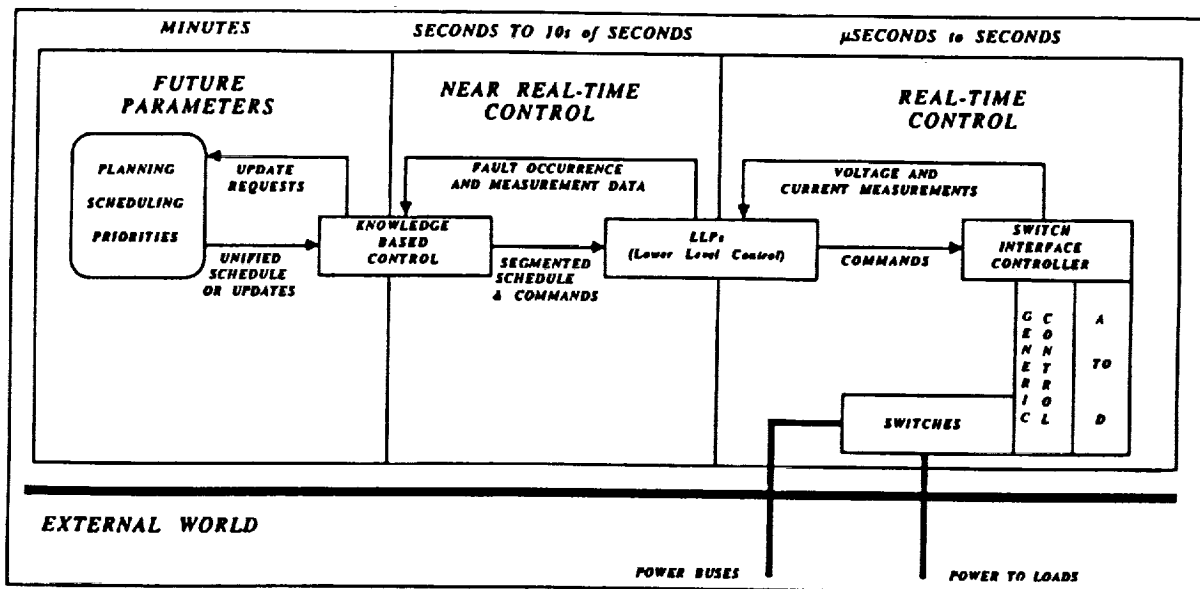


Figure 10 SSM/PMAD Control Flow

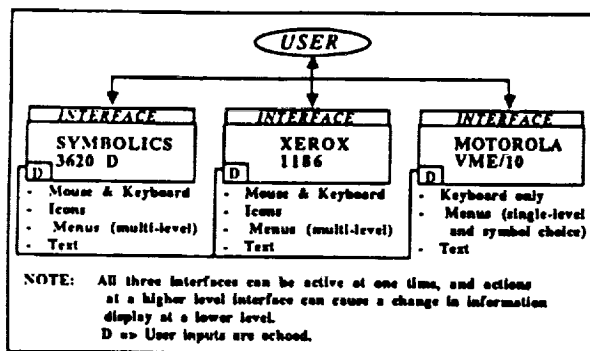


Figure 11 SSM/PMAD Breadboard User Interfaces

4. Concluding Remarks

The SSM/PMAD test bed possesses a rich and complex architecture which provides autonomous management capability. Our approach to its development was to view the entire problem and allocate appropriate technologies in a systems approach, rather than to try to isolate a domain specific problem around which to form a single knowledge base in a mundane approach. Its development, however, was not without a frustrating manifold of obstacles. Concurrent development of the switchgear hardware and the automation software [6] gave rise to development weaknesses in the FRAMES knowledge base (who can be an expert on hardware which does not yet exist?) and caused extra iterations in the complete development cycle. This problem has been overcome, but experience is what you get when you don't have any.

The system is now being examined for directions in growth. Growth is one of the key *plus* features of knowledge bases, but it is also not without its negative potential. When this project started in 1985 the hardware chosen was adequate for the plans and requirements which then existed. Those attributes grew during the initial three years of development until now they are larger than the capability of the processors. As new functionality is added to the overall system each component degrades slightly in performance. Questions then arise. How is performance improved? And, which technological area is the weakest (needing added strength the most)? Generally, the findings have been that the knowledge

based areas are in greatest need of improvement in terms of performance. Performance issues, however, not only relate to processor speed but also to how the knowledge is both organized and accessed. The FRAMES portion of the SSM/PMAD test bed is undergoing improvement through the addition of a more complete knowledge base management system which integrates database management activities internally.

A new requirement we are examining is that of "intermediate levels" of autonomy implying the capability of a user to take over various levels of power within the system, such that the real-time understanding of the overall system and its topology are subject to change. This requires the introduction of a new knowledge based planning function which could instruct the other knowledge base functions on how to cope with these changes. Therefore, we are also planning to introduce new general purpose workstation hardware to the environment. This would provide processor performance levels as available today, yielding new and more exciting capability levels. The current capabilities of the SSM/PMAD system architecture will not diminish, but will remain strong and flexible.

5. Acknowledgement

This work was performed by Martin Marietta Astronautics Group under contract number NAS8-36433 to the NASA George C. Marshall Space Flight Center, Huntsville, Alabama.

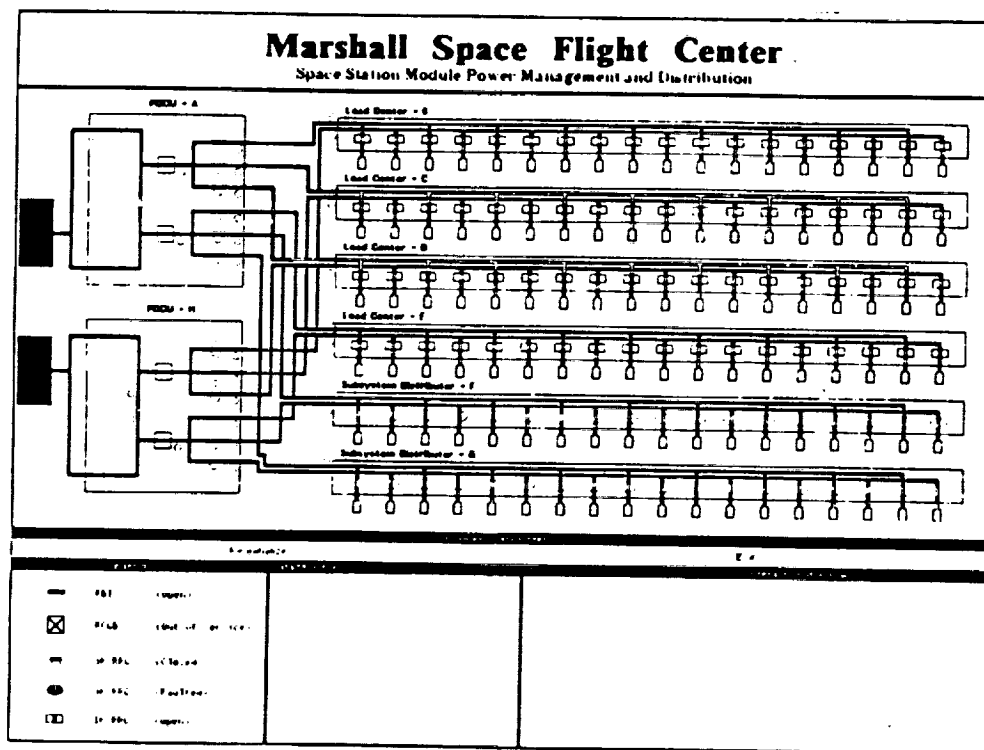


Figure 12 The SSM/PMAD FRAMES User Interface

6. References

- [1] P. Anderson, J. Martin, and C. Thomason, "Automated Power Distribution Hardware", Proceedings of the IECEC, 1989.
- [2] D.L. Britt, J.R. Gohring, and A.L. Geoffroy, "The Impact of the Utility Power System Concept on Spacecraft Activity Scheduling", Proceedings of the IECEC, page 621, 1988.
- [3] B. Chandrasekaran, "Towards a Functional Architecture for Intelligence Based on Generic Information Processing Tasks", Proceedings of the International Joint Conference on Artificial Intelligence, page 1183, 1987.
- [4] J.L. Dolce and K.A. Faymon, "A Systems Engineering Approach to Automated Failure Cause Diagnosis in Space Power Systems", Proceedings of the IECEC, page 590, 1987.
- [5] P. Fink, J. Lusth, and J. Duran, "A General Expert System Design for Diagnostic Problem Solving", IEEE Transactions on Pattern Analysis and Machine Intelligence, page 553, September 1985.
- [6] K.A. Freeman, R. Walsh, D.J. Weeks, "Concurrent Development of Fault Management Hardware and Software in the SSM/PMAD", Proceedings of the IECEC, page 307, 1988.
- [7] M.R. Genesereth, "An Overview of Meta-Level Architecture", Proceedings of the National Conference on Artificial Intelligence, page 119, 1983.
- [8] M. Georgeff and O. Firschein, "Expert Systems for Space Station Automation", IEEE Control Systems, page 3, November 1985.
- [9] A. Gupta and M. Tambe, "Suitability of Message Passing Computers for Implementing Production Systems", Proceedings of the National Conference on Artificial Intelligence, page 687, 1988.
- [10] M.A. Kelly and R.E. Seviara, "A Multiprocessor Architecture for Production System Matching", Proceedings of the National Conference on Artificial Intelligence, page 36, 1987.
- [11] P.J. Kline and S.B. Dolins, "Choosing Architectures for Expert Systems", Rome Air Development Center, Air Force Systems Command, Griffis AFB, RADC-TR-85-192, October 1985.
- [12] L.F. Lollar and D.J. Weeks, "The Autonomously Managed Power Systems Laboratory", Proceedings of the IECEC, page 415, 1988.
- [13] W. Miller, et al, "Space Station Automation of Common Module Power Management and Distribution", Martin Marietta Aerospace Denver Astronautics Group, 1989.
- [14] W.D. Miller and E.F. Jones, "Automated Power Management Within a Space Station Module", Proceedings of the IECEC, page 395, 1988.
- [15] W.D. Miller and E.F. Jones, "Automated Space Power Distribution and Load Management", Proceedings of the IECEC, page 544, 1987.
- [16] NASA, "Space Station Advanced Automation Study Final Report", Strategic Plans and Programs Division, Office of Space Station, NASA Headquarters, May 1988.
- [17] R. Reiter, "A Theory of Diagnosis From First Principles", Artificial Intelligence, 32(1), page 57, 1987.
- [18] J. Riedesel, "A Survey of Fault Diagnosis Technology", Proceedings of the IECEC, 1989.
- [19] D.P. Siewiorek, C.G. Bell, and A. Newell, "Computer Structures: Principles and Examples", McGraw-Hill, page 332, 1982.
- [20] D.J. Weeks, "Space Power System Automation Approaches at the George C. Marshall Space Flight Center", Proceedings of the IECEC, page 538, 1987.
- [21] D.J. Weeks, "Expert Systems in Space", IEEE Potentials, Vol. 6, No. 2, 1987.

ORIGINAL PAGE IS
OF POOR QUALITY

REACTIVE AUTONOMOUS PLANNING IN SPACECRAFT

A PAPER PRESENTED AT THE AAAIC IN OCTOBER 1989.

REACTIVE AUTONOMOUS PLANNING IN SPACECRAFT

Barry R. Ashworth
Martin Marietta Astronautics Group
Denver, Colorado

ABSTRACT

Martin Marietta, under contract to NASA, Marshall Space Flight Center since 1985, is developing technologies and methodologies for use in the automation of power management and distribution subsystems. Often, these same methods apply to other spacecraft subsystems, such as attitude control and command and data handling. Autonomous management of subsystems includes the functions of monitoring, control, and diagnosis. In order to effectively control subsystem activities during the occurrence of faults, diagnosis alone is not sufficient. A plan of recovery reacting to the system state, the fault and its implications, and the desired goals is also needed.

Reactive planning introduces issues and concerns for successful autonomous functioning of spacecraft. This is especially true when various subsystem interactions are considered while diagnosing faults and analyzing trends within the overall system. Power distribution and control is especially sensitive to faults and interruptions, as other subsystem activities may be interrupted and overall system catastrophic failure may result. In this paper an approach to reactive planning used in power subsystem management is presented. Subsystem interaction issues which effect overall system activities and performance are examined. A knowledge-based system to implement power subsystem reactive planning is discussed along with its associated implications to autonomy. Interactions between control, diagnosis, and planning functions are also considered.

1. Nomenclature and Symbology

AI	Artificial Intelligence
AC, ac	Alternating Current
ADC	Analog to Digital Card
CAC	Communications and Algorithmic Controller
CAS	Communications and Algorithmic Software
CC(E)	Computation and Control (Engine)
DC, dc	Direct current
FELES	Front End Load Enable Scheduler
FRAMES	Fault Recovery and Management Expert System
GC	Generic Controller
Hz, hz	Hertz (cycles per second)
K, k	Kilo (1000)
KW, kw	Kilowatt (1000 watts)
LC	Load Center
LLF	Lowest Level Function
LLP	Lowest Level Processor
LPLMS	Load Priority List Management System
MSFC	(George C.) Marshall Space Flight Center
NASA	National Aeronautics and Space Administration
PDCU	Power Distribution Control Unit

RBI	Remote Bus Isolator
RCCB	Remotely Controlled Circuit Breaker
RPC	Remote Power Controller
SIC	Switch Interface Controller
SSM/PMAD	Space Station Module (Autonomous) Power Management and Distribution

○ SENSOR	⊥ REMOTE	□ REMOTE	◆ REMOTE
RMS VOLTAGE	POWER	CONTROLLED	BUS
RMS CURRENT	CONTROLLER	CIRCUIT BREAKER	ISOLATOR
POWER FACTOR	1 OR 3 KW	10 KW	15 KW

2. Introduction

"During Magellan's 15-month cruise to Venus, other than periods of high activity, the spacecraft team goes through a daily ritual of *monitoring* the spacecraft, *analyzing* telemetry, *predicting* performance, and *planning* for upcoming events."

"On a typical day, spacecraft team engineers in Denver first check the status of the spacecraft's six major subsystems: command and data, power, attitude control, telecommunications, propulsion, and thermal."[7]

Spacecraft operations management consistently contends with situations of the type described above. The planning activities center around three fundamental drivers: first, the health of the subsystem being examined; second, the tasks that need to be accomplished by the integrated subsystems actions; and third, the level of priorities assigned to each of the needed tasks and how those priorities are managed. Another important consideration in spacecraft activities planning is the overall level of intelligent and adaptive control. If spacecraft subsystems can be regarded as autonomous, the planning activity involving them becomes somewhat more tractable due to a greater degree of predictability within a closed world representation for the planner (e.g., subsystems as described in [9,12]).

A planning activity configuration for a spacecraft subsystem is shown in Figure 1. The planning involved is *reactive* due to needed subsystem and environmental interactions. The reactive planning functions respond to commanded inputs, anomalies or faults within the complete system, and how well in general the overall system is executing its current plan. Needed changes are made either updating the current plan or producing an entirely new plan if the present one is unmanageable.

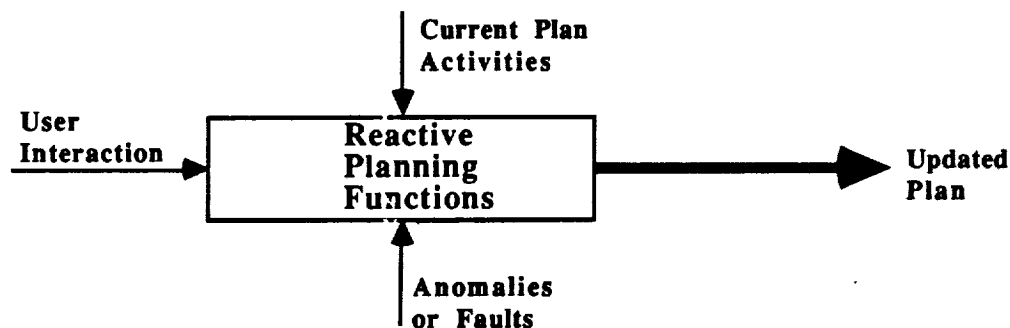


Figure 1 System Planning Activity

3. Spacecraft Autonomy and Execution

3.1 The SSM/PMAD Architecture

The breadboard structure for the SSM/PMAD at NASA/MSFC is shown in Figure 2. The objective of the breadboard operation is to provide autonomous management of module internal power needs in a Space Station Freedom similar structure [2,8,13,14].

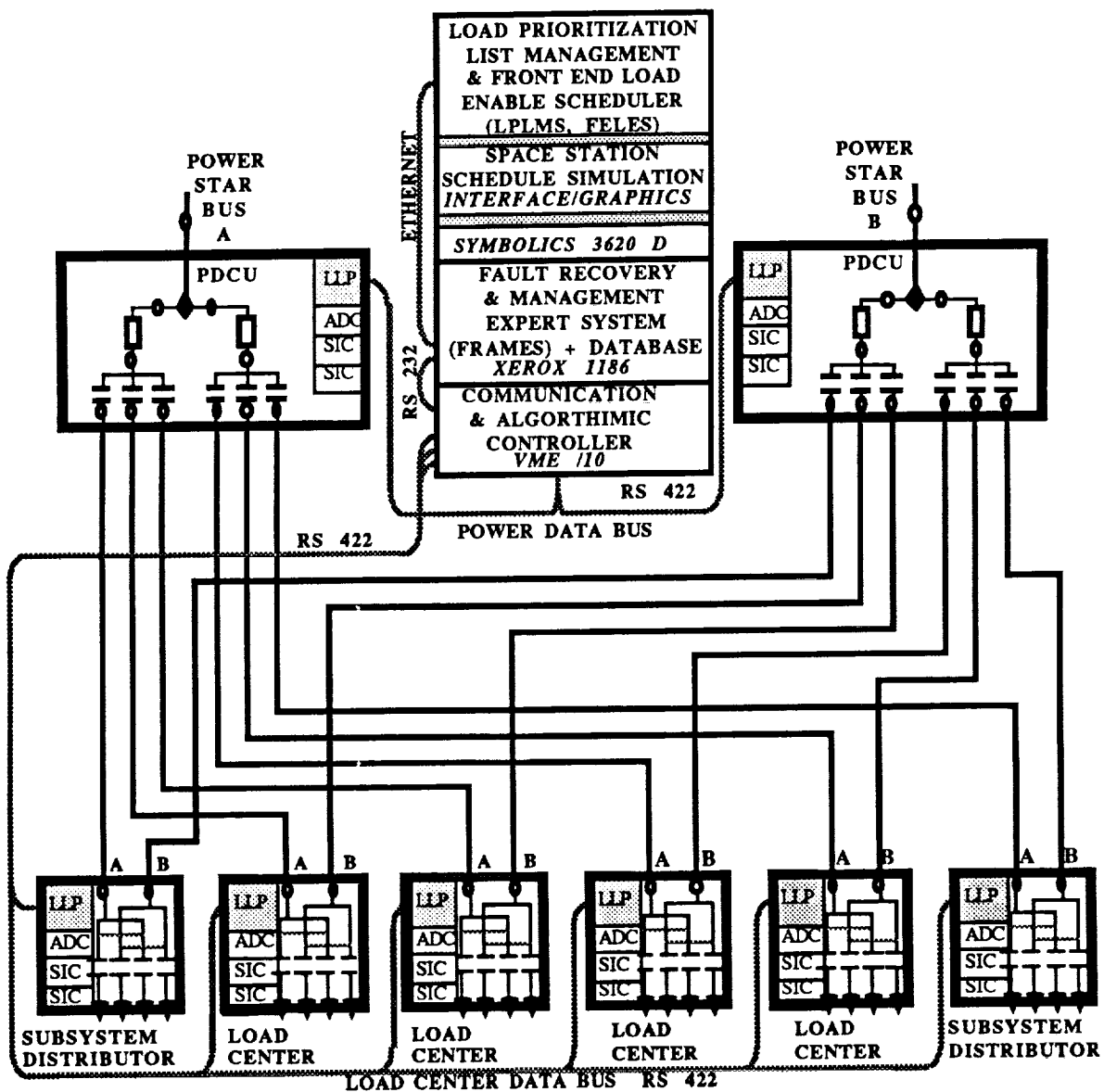


Figure 2 The SSM/PMAD Structure

A necessary function for providing the autonomous management is that of reactive planning. Planning in the sense of the breadboard is part of the high-level control activity. In fact, the entire flow of SSM/PMAD activity is one of intelligent control [11], embedding multi-agent AI systems for planning, scheduling, priority management, diagnosis, and even low-level management. The location of the control functions and how planning is related is shown in Figure 3.

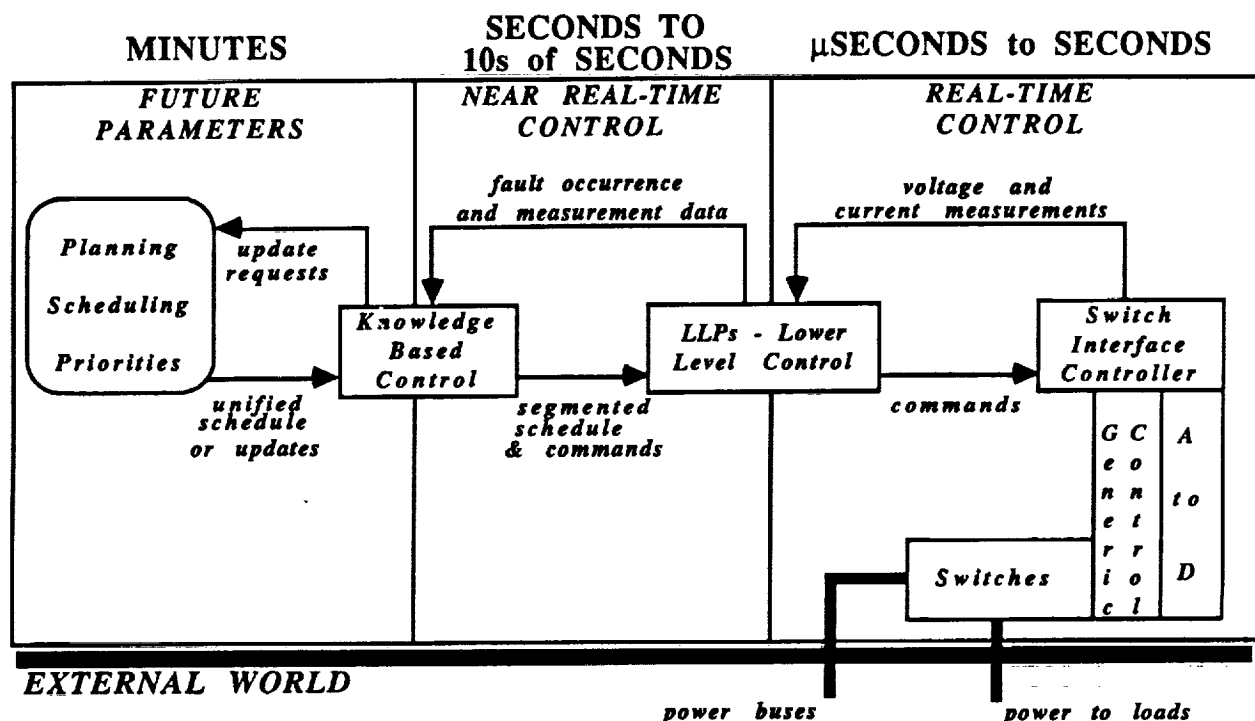


Figure 3 Planning in the SSM/PMAD Control Flow

As can be seen in the above figure, planning resides at the highest level. Monitoring of the actual power hardware occurs at the lowest level. The system has been configured in both 120 V dc and 220 KHz ac power versions. A CAC handles communications flow between the lower level and the knowledge based control activities. The present LLP CCEs are Motorola 68010 executing PASCAL based control programs, while all knowledge based planning, scheduling, and control functions execute on Xerox and Symbolics AI workstations utilizing their native LISP environments.

3.2 Reactivity in the SSM/PMAD Planning Activity.

The SSM/PMAD system must plan ahead to produce an initial schedule of events. This is imperative but does not preclude the need to react to the real-time stimuli which necessarily occur, such as anomalies, faults, or early or late event terminations. Therefore, the system must be one which is of the type that can plan, react, and replan, such as a robot but not necessarily with mobility [15]. Presently, the SSM/PMAD planning is achieved by human interaction with autonomy immediately following. An autonomous planning capability implementation is presently being examined from several aspects.

First, planning must be successfully integrated with the scheduling mechanism. In order for a planning activity to exist within the SSM/PMAD it must maintain the system run-time autonomy capability. Therefore, reactive replanning must accept the schedule constraints provided by the suite of FELES scheduling activities autonomously during system execution [1,3,10]. It is possible to establish more than one plan which is conformable to multiple schedules and vice versa. Therefore, any needed conflict resolution is achievable through maintaining a set of on-line, run-time heuristics which mitigate the conflicting situations.

Second, conjunctive plan goals must be achievable. The real world consists of many complex activities to be managed, each of which is capable of levying at least one goal on a planning activity [4]. Therefore, the autonomous SSM/PMAD planning function allows for the achievement of both subgoals and primary goals with interrelationships allowed as separately maintained heuristics. For example:

*enable switch 11 at time period 5
and switch 12 at time period 5
and switch 16 at time periods 5 and 6
and switch 21 at time period 6*

may only be achievable if the load at switch 21 is variably constrained at enable-time. So, the planner would need to establish the constraint and communicate it successfully to the overall system and possibly even the user interface.

Third, the run-time system must be predictable to the planner, and the planner must be predictable and controllable from the human user domain. Therefore, the planner must share in the model knowledge of the system domain and must react to the anomalies, faults, and user requests, *such as partial system autonomy* [5,6]. This allows for the system suppleness and conformity which often disappears whenever autonomy or high levels of automation are introduced.

In order to implement the planner described, it is necessary to be able to manage multiple independent knowledge bases concurrently. Presently implemented in the SSM/PMAD system is a knowledge base management system which provides that capability. This allows for the proper order of reasoning in the application of event processing and reaction. Therefore, the occurrence of events in the system sets in motion the correct knowledge relations which lead to an appropriate replan and reschedule. For example, a user requesting manual control of a suite of switches would only cause those switches and their successors to be affected at the lowest level priority as determined by the LPLMS, relieving higher level priority activities from being affected. *This provides one of the key advantages to reactive autonomous planning: the capability to return automated features of the overall spacecraft system to absolute human control, a little at a time.*

4.0 Concluding Remarks

Many areas within spacecraft system environments can benefit from reactive autonomous planning. Every subsystem which interacts, either directly or indirectly, with other subsystems could benefit from at least an off-line monitoring application including reactive autonomous planning. This would help alert operations personnel as to conflicts which may arise based upon potential commanding or scheduling work-arounds during fault periods. However, the greatest gain could be made by actually employing these planning systems as part of the on-board capability package, yielding more intelligent

spacecraft capable of tending to themselves during periods when human interaction is unavailable or undesirable.

Lower long-term operations costs are achievable for the Space Station Freedom employing reactive autonomous planning. This is also true for other complex space operational environments as the need for increased levels of personnel is diminished. Also, a spacecraft's safety and operability are increased as problems become recognized before they are introduced to the system, and faults are handled while the danger level is minimal. Automated planning features are now more reasonable as more and more processing power for space systems is becoming available (the 80386 Intel processor for the Space Station Freedom), and general purpose processing environments can now handle complex AI problem solutions.

5. Acknowledgement

This work was performed by Martin Marietta Astronautics Group under contract number NAS8-36433 to the NASA George C. Marshall Space Flight Center, Huntsville, Alabama.

6. References

- [1] J.A. Ambrose-Ingerson and S. Steel, "Integrated Planning, Execution and Monitoring", *Proceedings of the AAAI*¹, page 83, 1988.
- [2] B.R. Ashworth, "An Architecture for Automated Fault Diagnosis", *Proceedings of the IECEC*², page 195, 1989.
- [3] D.L. Britt, J.R. Gohring, and A.L. Geoffroy, "The Impact of the Utility Power System Concept on Spacecraft Activity Scheduling", *Proceedings of the IECEC*, page 621, 1988.
- [4] D. Chapman, "Planning for Conjunctive Goals", *Artificial Intelligence*, 32, page 333, 1987.
- [5] E.H. Durfee and V.R. Lesser, "Predictability Versus Responsiveness: Coordinating Problem Solvers in Dynamic Domains", *Proceedings of the AAAI*, page 66, 1988.
- [6] R.T. Hartley, M.J. Coombs, and E. Dietrich, "An Algorithm for Open-World Reasoning Using Model Generation", *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, page 193, 1987.
- [7] Martin Marietta Corporation, "Magellan Mission Bulletin", June 9, 1988.
- [8] W.D. Miller, et al, "Space Station Automation of Common Module Power Management and Distribution", Martin Marietta Aerospace Denver Astronautics Group, 1989.
- [9] NASA, "Space Station Advanced Automation Study Final Report", Strategic Plans and Programs Division, Office of Space Station, NASA Headquarters, May 1988.
- [10] P.S. Ow, S.F. Smith, and A. Thiriez, "Reactive Plan Revision", *Proceedings of the AAAI*, page 77, 1988.
- [11] J. R. Riedesel, C.J. Myers, and B.R. Ashworth, "Intelligent Space Power Automation", *Proceedings of the IEEE International Symposium on Intelligent Control*, 1989.
- [12] T.M. Trumble, "Scoping Array Automation", *Proceedings of the IECEC*, page 201, 1989.
- [13] B.K. Walls, "Exercise of the SSM/PMAD Breadboard", *Proceedings of the IECEC*, page 189, 1989.
- [14] D.J. Weeks, "Space Power System Automation Approaches at the George C. Marshall Space Flight Center", *Proceedings of the IECEC*, page 538, 1987.
- [15] D.E. Wilkins, "Practical Planning: Extending the Classical AI Planning Paradigm", Morgan Kaufmann Publishers Incorporated, © 1988.

¹ AAAI - American Association for Artificial Intelligence.

² IECEC - Intersociety Energy Conversion Engineering Conference (IEEE, AIChE, ANS, SAE, ACS, AIAA, ASME).

**AUTONOMOUS OPERATION OF A SPACE STATION FREEDOM
TYPE POWER TESTBED**

A PAPER PRESENTED AT THE NASA/JSC FAULT DIAGNOSIS WORKSHOP IN JUNE 1990.

Autonomous Operation of a Space Station Freedom Type Power Testbed

Barry Ashworth
Martin Marietta Astronautics Group
Denver, Colorado

Bryan Walls
NASA, George C. Marshall Space Flight Center
Huntsville, Alabama

ABSTRACT

Keywords: Fault Diagnosis, Power Autonomy Testbed, Space Power

Scheduling and monitoring spacecraft power systems has traditionally required intensive ground support. In general, ground support activities for power management consist of managing the planning, scheduling, and distribution of power to required loads; diagnosis of power system faults; and control of fault recovery. Martin Marietta, under contract since 1985 to NASA, George C. Marshall Space Flight Center, has designed and implemented a power system testbed for Space Station modules which automates power management and distribution as well as power system fault diagnosis and recovery.

PROBLEM DEFINITION

Reduction of ground support costs for complex space systems, such as the Space Station Freedom, is needed but not easily accomplished. Human expert support is a must in the operation of these extremely complex environments. Often, situations arise which require reallocation of resources already in use or scheduled to be used. These situations almost always appear when faults occur in spacecraft subsystems. One subsystem particularly sensitive to the reallocation syndrome is the Power Subsystem. This is for two reasons. First, power affects the operations of almost all other activities on a spacecraft, including life support, attitude control, and thermal management. Second, there is always more demand for power in a complex space environment than is available. This high demand, with the corresponding requirement for a large number of people working around the clock to plan and schedule the use of the power, presents a large cost in complex space systems operations. The problem is, then, how to go about reducing that operational cost.

USE OF AI TECHNOLOGY IN THE SSM/PMAD TESTBED

For purposes of the Space Station Module Power Management and Distribution (SSM/PMAD) program the problem's solution was broken down into several parts and was given unique direction for operation. The parts were Power Management and Distribution, Priority Management, Scheduling, Fault Management and Recovery, and Low Level Deterministic Control. The unique direction was to provide an automated testbed in which to implement the solution parts, and for that testbed to ultimately provide hypothesis testing capability concerning power system design and management. Knowledge based solutions were employed in all SSM/PMAD components except in the low level deterministic control. In this paper we will examine the advanced automation concepts of the SSM/PMAD testbed, and how automated fault diagnosis and recovery are utilized.

POWER MANAGEMENT AND DISTRIBUTION AUTOMATION

The goals of reducing ground support costs and increasing feasibility of long term, long distance missions motivates the automation of space power management. The Space Station module Electrical Power System (EPS) requires the management of power to as many as 88 distribution locations, each with up to a dozen loads – larger than anything yet flown [3]. This complexity introduces difficult issues in both hardware and software control within the overall environment.

In automating the SSM/PMAD testbed, four primary and necessary tasks were identified. These were: 1) planning and resource allocation, 2) scheduling of power to particular switches for operation of loads, 3) management of load priorities, and 4) fault diagnosis and recovery. Each of these represents a separate AI knowledge agent, and three of them are now operational and deployed in the SSM/PMAD testbed at NASA/MSFC.

PLANNING AND SCHEDULING

The planning and scheduling problems can be operationally intensive tasks. Witness the example of SKYLAB. Mission operations had to be scheduled (and sometimes planned, especially within contingencies) in real time by a large ground support task force. The result was up to 20 ground crew working with a flight crew of three to handle EPS operations. Automation of the planning and scheduling tasks, especially in response to changing power availability, becomes increasingly necessary when considering 2 to 3 times the human involvement required for Space Station Freedom (IOC) [3].

In general, the planning and scheduling problems are NP-complete problems (the number of possible solutions are exponential with respect to the number of activities and the available resources) requiring the skilled use of heuristics to manage them. The scheduling problem is to optimize a set of tasks with respect to efficiency within a given set of temporal constraints. If every possible solution were to be analyzed the problem might never be solved.

In the domain of spacecraft power systems the planning/scheduling problem requires a large amount of knowledge about the loads being scheduled. Loads have diverse and changing energy requirements. Loads may also be constrained to operate in determined periods of time based on, for example, the visibility of certain star clusters; on the sensitivity of the load itself (e.g. does it require a minimal amount of vibration to operate correctly); and even on the availability of crew members to operate the loads in a required manner.

LOAD PRIORITY MANAGEMENT

Loads also have priorities, life support systems being one of the most critical. Management of these priorities is an important function. The necessity for a load to stay active may change in time due to both external events and to maturing functions internal to the load (a heater bringing process elements to a sufficient operating temperature, for example). As the priority landscape changes, the interrelations of the overall system may also change, requiring an appropriate management response to these changes.

FAULT DIAGNOSIS AND RECOVERY

In the event of a fault in the power system, automation of the diagnosis of the fault is required if power management autonomy is the goal. The process of diagnosing a fault in the power system and scheduling around the failed parts of the system defines a mode of fault recovery.

The primary motivation of automated fault recovery is to continue operation with minimal interruption of the loads. By scheduling around the affected parts of the power system it becomes possible to manually repair the affected regions of the power system for subsequent use, without shutting down the entire system.

There may be many different faults occurring in a power system. These include a single hard fault detected by a switch tripping (this is usually caused by an open or a low resistivity short within a circuit). If a switch is not operating correctly a masked fault may occur where the switch does not trip but the one above it (closer to the source) does. Fault diagnosis is also complicated by multiple independent faults occurring in the same time period. The diagnosis of faults depends to some degree on both the placement and accuracy of sensors for fault detection as well as the ability to manipulate switches in the power system in an attempt to isolate the location of the fault and provide confirming evidence of a particular fault.

Manipulation of switches in the power system domain as a fault isolation technique is dependent on the hierarchical nature of the switches. This is guaranteed by the acyclic nature of the control for power systems (where power flows from the source to the loads, guided by switches).

LOW LEVEL DETERMINISTIC CONTROL

The actual control of the power switchgear is provided by deterministic software. An important function of the deterministic control element is to act as the intermediary between the power management knowledge agents and the power switching hardware. This level of software quickly accomplishes those activities which are at a higher level than can be accomplished by the switchgear, but doesn't require the "thinking power" of higher level systems. Normal switching at the lowest level is guided by a schedule of events. Redundant power bus sourcing is provided at this level, as are load sheds for violation of power limitations.

THE USER INTERFACE

Automated systems require very capable user interface functionality. In using the SSM/PMAD, a user may engage its services to assume manual control or collect data at any time. Therefore, the user interface must have access to all functionality which may be affected by a user's actions. In providing this capability, it is important not to require the user to possess detailed knowledge of the entire suite of system functionality. This requirement makes the user interface for the SSM/PMAD an intelligent service which integrates its services with those activities carried on by the other functions within the system.

DESCRIPTION OF THE APPLICATION

The deployed SSM/PMAD testbed may be described at a number of levels. At the bottom level is the switchgear hardware while at the top level scheduling, priority management, and fault diagnosis reside. In between these levels exists software and hardware algorithms designed to further enable the automation process, supporting the higher processes. We will describe the supporting hardware and software of the testbed and then talk further about the scheduling and fault diagnosis software. Figure 1 shows the topology for the 120 V dc version of the SSM/PMAD testbed, which was made operational at NASA/MSFC in October 1989 (a complete 20 kHz, 208 V ac version, including all knowledge agents became operational in December, 1988). The workstation Knowledge Base Management System (KBMS) software, KNOMAD, depicted in Figure 1 is to be installed in June 1990 and will provide parallel knowledge agent control.

THE POWER SYSTEM

The current implementation defines eight lowest level processor (LLP) units on which resides the low level deterministic software. Each LLP interfaces to two switchgear interface cards (SIC), which in turn interface to one analog to digital card (ADC) and as many generic controller cards (GC) as there are physical switches. The current implementation defines up to 18 switches being controlled per LLP.

From the perspective of power system automation there are two features of the physical power system to be noticed. The first feature is the architecture; the power system is laid out as a distributed system. Each LLP controls a set of switches. If one LLP fails the other LLPs are not affected (in general). In the current implementation there are two special LLPs that control distribution of power to six lower level units, each with their own controlling LLP. This provides the hierarchy of switches. It is important to note that the LLPs are processors, so the power does not actually flow through them. They serve as controllers for the actual power distribution hardware (similar to a microprocessor controlling the flow of fuel to an automobile engine). The second feature is communication; the LLPs are used for communicating both switch commands (open and close) to the switches as well as switch and sensor information from hardware back to the fault management and recovery knowledge agent, FRAMES (Fault Recovery and Management Expert System).

SCHEDULING

Scheduling the activities needing power is the first step in the operation of power management and distribution. Scheduling is performed by the MAESTRO knowledge agent and is initiated by the user and subsequently performed whenever there is an unforeseen change in the power system. The goal of the

scheduling process is to make use of the available power in the most efficient manner possible, taking into consideration load dynamics and resource availability, as well as inter- and intra-task relationships.

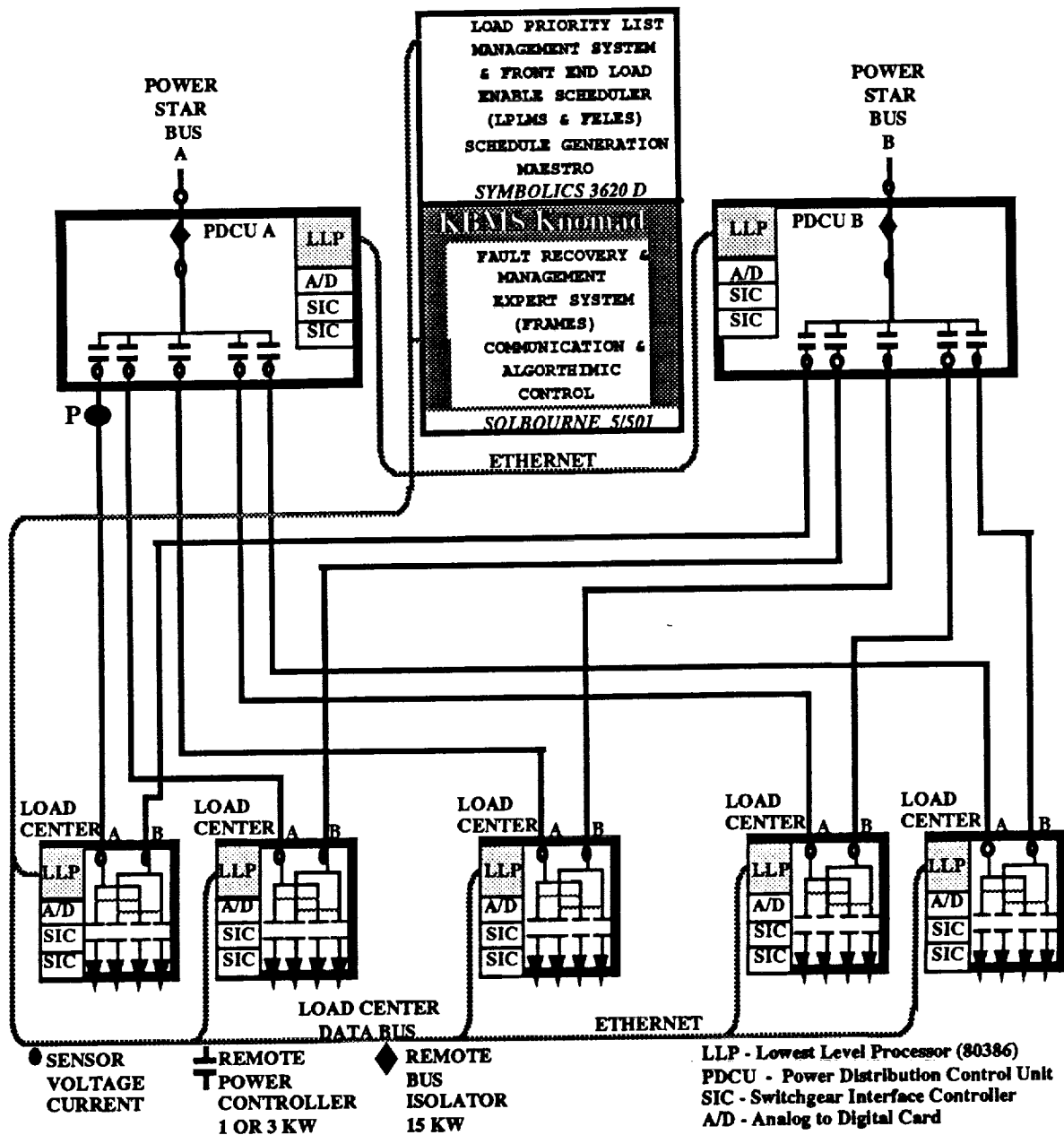


FIGURE 1 - POWER SYSTEM TOPOLOGY

When a fault occurs during operation, FRAMES diagnoses it and informs MAESTRO as to the identity of the switches which are no longer available for scheduling. Rescheduling resumes as many activities as possible and isolates the faulted areas until they are repaired. Rescheduling will also take place if the user specifies that there will be a change in the available power to the system for some future period of time.

FAULT RECOVERY AND MANAGEMENT

The purpose of FRAMES is to keep the power system operating as effectively as possible, especially in fault situations. This is done by using a model of the power system network, keeping track of both what is scheduled to happen and what is actually happening in the power system to recognize any discrepancies in the power system operation. When a discrepancy is detected FRAMES analyzes the situation to determine a fault that will account for the discrepancy.

The LLPs communicate various data about their switches and sensors to FRAMES. This includes amperage data, switch status information, and trip related information. When FRAMES discovers that there are symptoms indicating a fault in the power system, analysis determines what class of faults may account for the symptoms, and further testing may isolate the fault to a particular location in the power network. Once the fault location has been isolated as much as is possible (given available information from sensors and switches), FRAMES identifies to the scheduler those switches that are no longer useable. The scheduler uses this information to schedule loads on the remaining usable switches.

FRAMES, as managed within the KNOMAD environment, is designed to diagnose independent faults which either occur simultaneously in multiples or singularly. The types of faults FRAMES is designed to diagnose include hard faults, soft faults, masked faults, cascading faults, and incipient faults.

Hard faults are defined as faults that cause switches to physically trip. Soft faults are illegal uses of current in the power system, such as resistive shorts to ground. Incipient faults are situations that will become hard faults in a reasonably short time if nothing is done to avert them.

The user interface is attached as an internal knowledge driven FRAMES activity. This way, a user can gain access to the complete system structure through one common representation. A user can monitor the system or assume control at this single interface. The only other interface is provided to the scheduling agent for purposes of initialization.

THE AI DOMAIN OF THE SSM/PMAD

The primary application of AI that we are addressing is the automation of a power management environment utilizing multiple embedded knowledge agents (this does not consider the actual management of the multiple cooperating SSM/PMAD knowledge activities, which is a complete topic itself). This is done to greatly decrease the cost of developing and operating a very complex Space Station Freedom type module power management and distribution environment. The interaction of the knowledge agents with the real-time control elements of the system is quite innovative and presents extremely sophisticated and complicated problems with respect to information flow within the complete system. Figure 2 shows a logical flow of the information between the modules of the power system test bed.

The system distributes intelligence throughout the test bed as much as possible. It is made up of three artificial intelligence systems; the scheduler, the priority manager, and the fault management and recovery module. In addition, the LLPs contain rudimentary knowledge about power distribution and provide the low-level, activity driven, fast control of the switches. The LLPs must also contain rudimentary knowledge of what level of communication must take place with FRAMES in both nominal and fault situations.

The SSM/PMAD testbed may be used to test design hypotheses for the power distribution and control elements for the Space Station Freedom. This is done while minimizing human interaction with the overall process (e.g., a typical schedule for power consumption activities can be maintained and applied to more than one power consumption configuration) and allowing vigorous exercise of power components and loads. The high level AI embedded knowledge agents, running with the complete system in real-time, allow this. Otherwise, the SSM/PMAD would be just another large spacecraft-type power management and distribution system requiring manual processing and control at every juncture.

Multiple computational and reasoning agents cooperatively and autonomously manage the power system in both normal and fault situations. In particular, scheduling and priority management reside on a Symbolics LISP machine, the global level fault management and recovery system resides on a Solbourne 5/501 general purpose workstation, and the low level fault management and detection functions are in the lowest level 80386 Intel processors. These independent agents are responsible for their own areas of control, to respond to

situations as effectively as possible. The lowest level processors contain knowledge needed for fast response while operations at the scheduler may take more time without adversely affecting autonomous system operation.

The three AI systems interact such that when a hard fault occurs the power system is immediately protected by the smart switchgear in less than a microsecond. FRAMES recognizes the new configuration and decides if any other actions need to take place. FRAMES diagnoses the fault, recommends corrective action, and where appropriate may autonomously implement the corrective action. A determination is made if the current loads schedule has been perturbed by the anomaly and if so, the scheduler is directed to reschedule the loads for the remainder of the crew period. The priority manager then generates a new priority list about the loads which is downloaded to the LLPs. A similar sequence autonomously occurs in the event of a soft fault (except the switchgear doesn't trip) or when new directions or power allocation levels are sent down to the test bed (through the operator interface in the test bed). The test bed operator may also take manual control of the system at any time.

Figure 3 shows a basic diagram of the operation of FRAMES. FRAMES makes use of a model of the power system network (the switches, sensors, and cables) to determine both where faults may occur and how to further isolate faults.

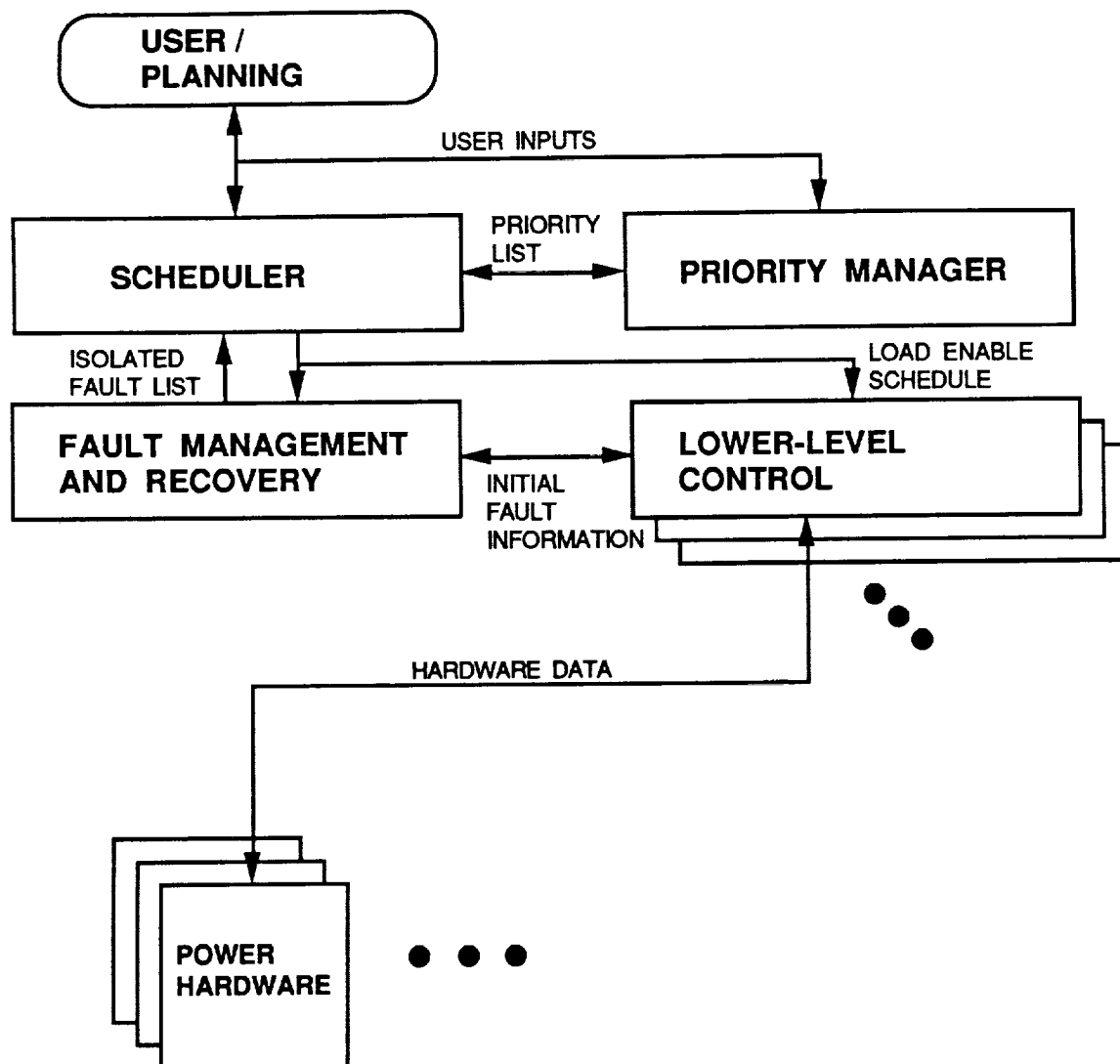


FIGURE 2 – SSM/PMAD LOGICAL INFORMATION FLOW

As data from the LLPs is sent to FRAMES, any tripped switches or violations in Kirchoff's Current Law indicate fault situations. Once fault situations are detected, further analysis is done and, depending upon the situation, manipulation of switches occurs to further isolate the fault and corroborate the diagnosis. For example, suppose there is a short below the 3kW switch (point P in figure 1). The 1kW switches below it will all trip on under voltage and the 3kW switch will trip on over current. This will indicate to FRAMES that first all the tripped switches should be opened (turned off). Next the 3kW switch is turned on and then, if it does not retrip, off. The action of turning the 3kW switch on trips it again. This indicates to FRAMES that the fault must be below the 3kW switch and above the 1kW switches.

CRITERIA FOR THE SUCCESSFUL SSM/PMAD TESTBED

There were two major goals that were achieved in this application of artificial intelligence to Space Station power automation. The first goal was autonomous operation. We have successfully incorporated autonomy into our application and tested it by injecting various faults and letting it recover and continue operation. This has also shown the effectiveness of our system by successfully incorporating knowledge, eliminating the need for large amounts of manpower to operate the SSM/PMAD.

The second goal was to design and implement distributed control of the system. This has also been successful. At the lowest level, each LLP autonomously controls its own switches, reporting data to the higher level agents. The power system can operate autonomously for periods of time even if the higher level agents should for some reason be unavailable.

An extra but very important feature of the SSM/PMAD testbed is its usability as a power system design test element which can take on major changes very rapidly. For example, a new topological design for the distribution of power can be incorporated into the SSM/PMAD operational knowledge agents via the

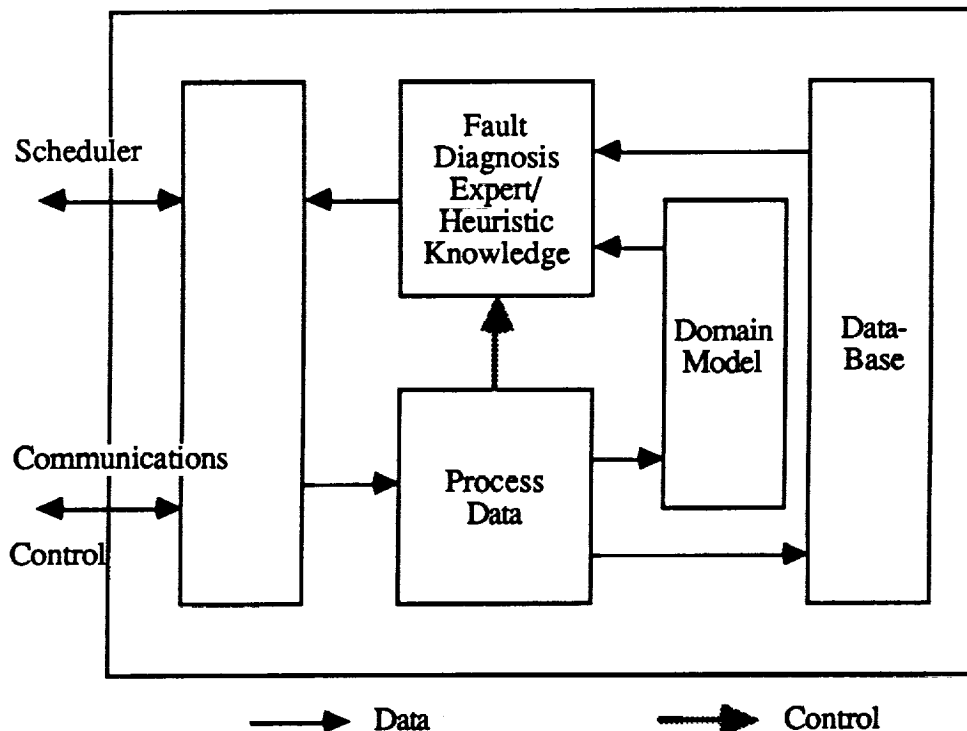


FIGURE 3 - FRAMES OPERATIONAL INTERACTION

FRAMES model representation and fault diagnosis rules. This provides NASA with important flexibility in analyzing actual power system componentry before deployment into a real spacecraft, giving strong cost savings by trapping development problems early.

FUTURE AND FURTHER ACTIVITIES

INTERMEDIATE AUTONOMOUS SYSTEM ACCESS

Intermediate levels of autonomous control is a necessary capability for the future SSM/PMAD testbed. This will allow a user to manually seize certain elements of the testbed while it continues automated operations with little or no impact. A system which is truly representational of an actual operational system should possess this characteristic.

PLANNING ACTIVITY

The knowledge agent aggregate must possess a planning capability unlike that which is used in scheduling. This is a necessary future enhancement for completion of the functionality of FRAMES during intermediate autonomous requests and operation. The planning agent will provide the necessary capability to review the existing testbed configuration, the user's goals, and the future needs of the automation function to produce a plan for minimal negative impact results.

SIMULTANEOUS MULTI-AGENT KNOWLEDGE MANAGEMENT

The SSM/PMAD possesses a simultaneous multi-agent knowledge manager function called KNOMAD (for KNOWledge MAnagement & Design). It utilizes a distributed database management function to provide a modified blackboard management capability [7]. This, along with the integrated capability of knowledge sharing through object-oriented frame representation, allows the multi-knowledge-agent concept to function within a real-time hardware environment. Growth of this system to include time domain dependencies is critical for successful Intermediate Autonomy requests.

DEMONSTRATION VIDEO TAPE

A demonstration video tape of the SSM/PMAD testbed at the NASA, George C. Marshall Space Flight Center is currently in production and will be available in July of 1990. The video will demonstrate the autonomous operation of the system, even during the occurrence of faults.

REFERENCES

- [1] Barry R. Ashworth. An Architecture for Automated Fault Diagnosis. In *Proceedings of the 24th Intersociety Energy Conversion Engineering Conference*, 1989.
- [2] Barry R. Ashworth. Reactive Autonomous Planning in Spacecraft. In *Proceedings of the Aerospace Applications of Artificial Intelligence Conference*, 1989.
- [3] Robert T. Bechtel. MSFC EPS Automation; Skylab to Space Station (IOC) comparison chart, 1989.
- [4] William D. Miller, et al. Space Station Automation of Common Module Power Management and Distribution - Interim Final Report. *NASA Contractor Report 4260*, November, 1989.
- [5] Joel D. Riedesel. A Survey of Fault Diagnosis Technology. In *Proceedings of the 24th Intersociety Energy Conversion Engineering Conference*, 1989.
- [6] Joel D. Riedesel, Chris J. Myers, and Barry R. Ashworth. Intelligent Space Power Automation. In *Proceedings of the Fourth IEEE International Symposium on Intelligent Control*, 1989.
- [7] Joel D. Riedesel. Knowledge Management: An Abstraction of Knowledge Base and Database Management Systems. *NASA Contractor Report 4273*, January, 1990.
- [8] Bryan K. Walls. Exercise of the SSM/PMAD Breadboard. In *Proceedings of the 24th Intersociety Energy Conversion Engineering Conference*, 1989.

A SURVEY OF FAULT DIAGNOSIS TECHNOLOGY

A PAPER PRESENTED AT THE IECEC IN AUGUST 1989.

A SURVEY OF FAULT DIAGNOSIS TECHNOLOGY

Joel Riedesel
Martin Marietta Astronautics
Denver, Colorado

ABSTRACT

Power system automation requires intelligent methods for diagnosing faults that may occur in the system. This paper surveys existing techniques and methodologies available for fault diagnosis. The techniques surveyed run the gamut from theoretical artificial intelligence work to conventional software engineering applications. They are shown to define a spectrum of implementation alternatives where trade-offs determine their position on the spectrum. Various trade-offs include execution time limitations and memory requirements of the algorithms as well as their effectiveness in addressing the fault diagnosis problem.

Martin Marietta, under contract since 1985 to NASA, Marshall Space Flight Center, is continuing the development of technologies and methodologies for use in the automation of power management and distribution.

1. Introduction

The fault diagnosis problem must be considered in context. The usefulness of diagnosing faults is dependent upon how that information is used. This *information efficacy* is probably the most important criteria in determining how fault diagnosis technology should be used in a system.

The Power System Automation group at Martin Marietta has researched and developed a power system automation test bed, Space Station Module Power Management and Distribution (SSM/PMAD), delivered to NASA/MSFC in December 1988 [12]. This test bed has four major components: Scheduling software, fault diagnosis software, algorithmic software, and hardware (making up the switches and controllers). This test bed is used in an autonomous fashion. Activities are described to the scheduler and subsequently scheduled on particular switches at specific times. The scheduler takes into account numerous constraints regarding power availability, resource availability (crew, tools, etc.) and others. The schedule generated is used by the algorithmic software to command switches on and off and ensure proper power usage through those switches. If a fault occurs in the hardware the algorithmic software notifies the fault diagnosis software of the symptoms seen as a result of the fault. The fault diagnosis software is then responsible for isolating where the fault occurred and what the fault was. This may require further testing by commanding switches on and off to get more information. Once the fault has been isolated, the scheduler is notified of any switches that are no longer operable. The

scheduler then reschedules the activities to take advantage of the remaining switches in the breadboard.

Thus, in this context, fault diagnosis is used to autonomously recover from failures in the power system hardware. The specificity of the conclusions generated from fault diagnosis is very important. The more accurately a fault may be isolated the better the power system may be utilized.

Consider figure 1. This figure shows power entering a Remote Controlled Circuit Breaker (RCCB) that can switch 15k watts. The output of the RCCB feeds three 3k watt Remote Power Controllers (RPCs). Each 3k RPC feeds eight 1k RPCs. All of these switches may trip due to fast trip (F-T), over current, and under voltage (U-V). The fast trip occurs at 300% of power, the over current at 120% and the under voltage at zero voltage. A load may be powered beneath any 1k RPC. Figure 2 shows the symptoms resulting from a low impedance short injected at point A. Figure 3 shows the symptoms resulting from a low impedance short injected at point B with the assumption that the 1k RPC above it has a failed current sensor. As can be seen, the symptoms are identical. If the fault diagnosis software cannot distinguish between the faults in some way, all the switches from the 3k RPC down to the 1k RPCs below it will not be available for use. In figure 3 it is clearly the case that we only want to diagnose the single 1k RPC as not being available for use.

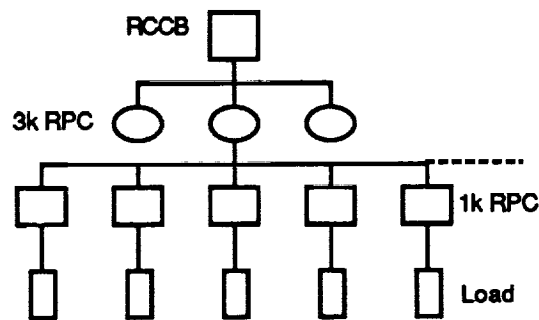


Figure 1

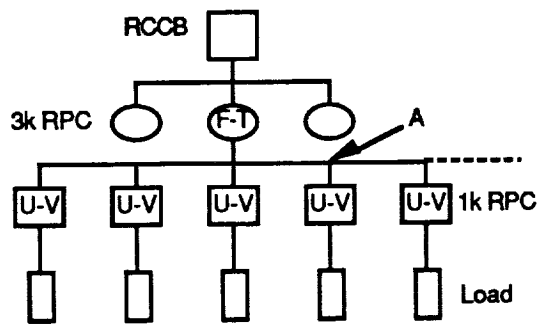


Figure 2

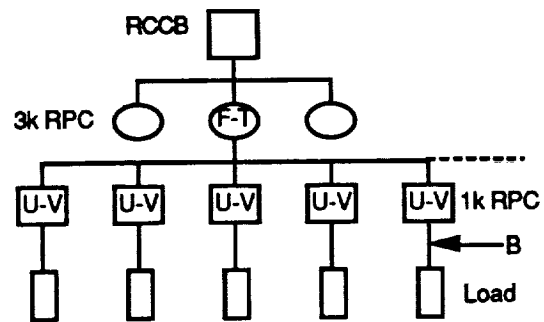


Figure 3

Section two defines the fault diagnosis problem in more detail. Section three surveys existing techniques and methodologies for solving the fault diagnosis problem. It outlines criteria used in comparing techniques as well as describing limitations of various alternatives. Finally, section four gives a brief conclusion.

2. The Fault Diagnosis Problem

When a fault occurs a set of symptoms occur. This was shown in figure 2. A fault is a one to many mapping to symptoms. The problem of fault diagnosis is to go in the reverse direction. Given a set of symptoms, what fault(s) account for them. This is a many to one mapping. Furthermore, different faults may share symptoms. In the case of the example above, the two different faults shared the symptoms exactly. Figures 4 and 5 are alternative ways of viewing this problem.

Shared Symptoms

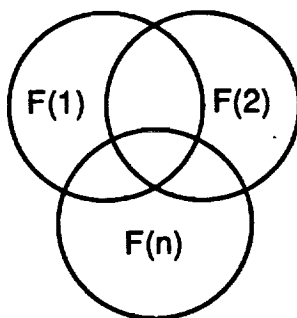


Figure 4

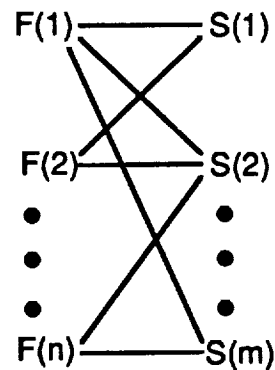


Figure 5

The fault diagnosis problem is complicated by the possibility of multiple faults occurring. In this case a single set of symptoms is given but no one fault describes the

symptoms sufficiently. The problem is now even more complex. Potentially, one could examine every combination of faults that could lead to the symptoms, a 2^n problem space where n is the number of faults possible. This naive approach would be ridiculous to implement though. Essentially this problem is a set-covering problem. The set is the set of symptoms. Faults are used to cover the set. There are pattern recognition algorithms in the field of machine learning that perform set-covering which could be applied here. Various heuristic approaches apply as well.

Whereas in single fault diagnosis it is possible to have a very high probability of correctness in the conclusion, in multiple fault diagnosis this probability has the potential of dropping dramatically.

Fault diagnosis is not a static problem. By this it is meant that fault diagnosis may dynamically perform various tests to isolate the fault. This updates the set of symptoms that are relevant to the fault(s). Thus, any algorithms that are defined ought to take into consideration this dynamic nature of the problem space. Furthermore, during diagnosis of a fault, another fault may occur. How do fault diagnosis algorithms update their symptom sets appropriately?

3. Techniques and Methodologies

In this section the fault diagnosis problem is given a very general model which is specialized by the various techniques and methodologies. The issues in fault diagnosis are described including how they fit into the model and how the techniques and methodologies are applied to them.

3.1. A Fault Diagnosis Model

Figure 6 depicts the model of fault diagnosis. In general, every fault diagnosis implementation consists of the parts in this model in some form. The box labeled *Model* is used to represent the *Artifact*. The use of the *Simulator* and artifact are to provide *Expectations* and *Data* to the *Problem Solver*. The problem solver consists of three parts; discrepancies between the expectations and data are first identified producing symptom sets, hypotheses are then generated based on the symptom sets, and finally, testing and probing are performed to discriminate between possible hypotheses and to confirm or corroborate conclusions.

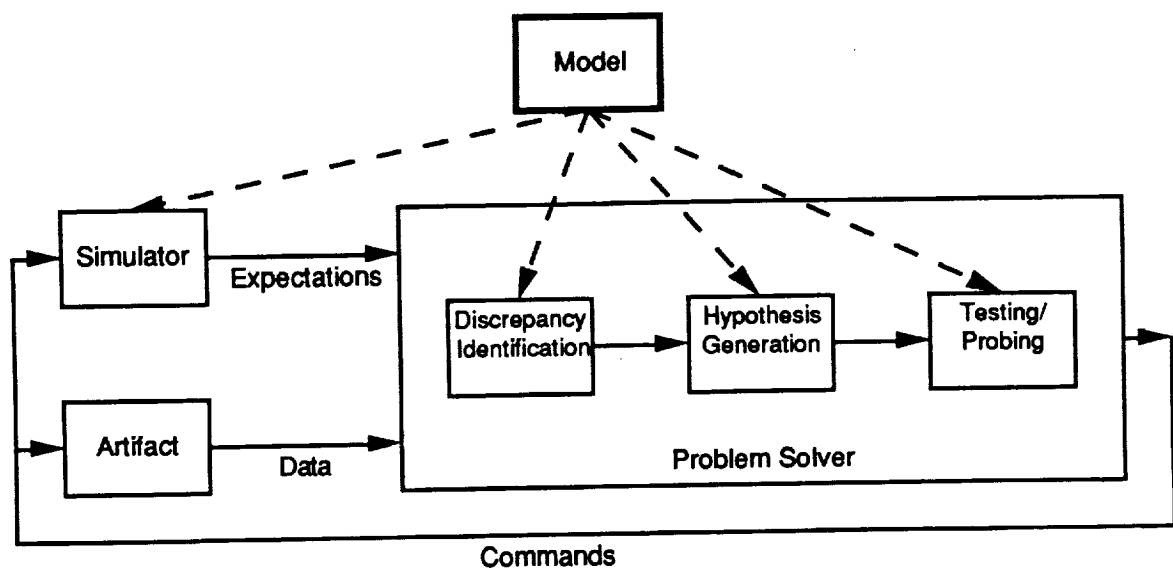


Figure 6

3.2. Issues in Fault Diagnosis

There are a number of issues in fault diagnosis. These issues relate to the alternatives available in their solution. Each alternative has implications on computation required by the CPU, space requirements, and their effectiveness in addressing the fault diagnosis problem. Briefly, these issues are:

- The Simulator: Qualitative vs. Quantitative
- Discrepancy Identification. Computation of Symptom Sets: Static vs. Dynamic
- Hypothesis Generation: Model-based, Rule-based, Completeness
- Single vs. Multiple Fault Identification

3.2.1. The Simulation Issue

The main question in simulation deals with the sufficiency of a qualitative model versus the necessity of a quantitative model. The trade-off in qualitative versus quantitative models is in the amount of processing necessary to generate an expectation. For the electronic circuit domain, there exist some good quantitative simulation tools. For other domains, such as a nuclear reactor, there are no general purpose simulators available. There are at least two reasons why a qualitative model may be desired. One, in the absence of robust, quantitative simulators, a qualitative simulator must suffice. And two,

qualitative simulators may be sufficient to provide the necessary expectations for the fault diagnosis problem, while at the same time economizing on the processing necessary to produce the expectations.

A model for simulation may be represented in many forms. The component-centered model [11] provides the most direct mapping between the model and the artifact as well as being more accessible to a user. The difficult problem in modelling is to represent the artifact in such a way that the possible discrepancies are readily identifiable. In multiple fault diagnosis, deKleer introduces a representation that combines the model and the discrepancy identification into one [10]. Other possibilities include analyzing the artifact to determine the information it can provide and building a qualitative model that corresponds to the information provided by the artifact.

The effectiveness of a fault diagnosis algorithm will be dependent upon the quality of a qualitative simulation. A quantitative simulation should be able to provide for a complete account of expectations, while a qualitative one may not be accurate enough.

3.2.2. The Discrepancy Identification Issue

Identifying the symptom sets of a fault is probably the easiest part of the fault diagnosis problem, although the computation of symptom sets has implications on CPU, memory, and effectiveness. A symptom set is derived from the difference between the expectations from the model and the data from the artifact. However, if the model is wrong, some symptom sets may be derived which the problem solver needs to recognize as a fault in the model.

The symptom set identified as discrepancies between the expectations and the data is then used and compared against the possible symptom sets. The computation of the possible symptom sets for comparison against the symptom set found is where the trade-offs become apparent.

Symptom sets may be computed statically by analyzing the artifact for all the possible faults and generating a table of symptoms and faults [12]. Symptom sets may also be computed statically by analyzing a component-centered model to produce a similar table. This allows the computer to generate the symptoms automatically. Drawbacks to this approach are based on the ability of the model to represent the artifact completely enough to produce all the symptoms and faults that the artifact would produce. This may require a quantitative model. Symptom sets may also be computed dynamically based on the model. As data is collected from the artifact and expectations are collected from the model, any discrepancies suggest that a symptom set should be computed. For example, in the

electronic circuit domain, if a discrepancy is noted, heuristics may be used to determine possible faults based on the discrepancy. Each of the possible faults is then plugged into the model to determine the symptoms that would result [15]. These are then compared with the observed symptoms from the artifact. Another method is to represent the dependencies between components in the model and when a discrepancy is detected, note all the possible components that could have faults, based on the dependencies, that could possibly have produced the discrepancy [10].

Static computation of symptom sets are more difficult initially and require more processing. They also require space to store all the symptom sets. Their advantage is that fault diagnosis processing may be speeded up. Dynamic computation of symptom sets requires CPU at the time of the fault but has little overhead in memory requirements. If a quantitative model is used dynamic computation may be very expensive in terms of CPU usage. On the other hand, if an artifact has many components, each of which has various types of faults, static computation may produce very bulky symptom and fault tables. One interesting possibility is to compute the symptom and fault tables up front and enter these into a machine learning program that will produce rules. These rules can then be used in an expert system by entering run-time symptoms to the program and getting a fault back. Incorporating testing capabilities into machine generated rules may be more difficult.

The completeness of the symptom sets is also an issue for effectiveness. If some symptom sets are not able to be computed or have not been determined to be possible then some faults may not be diagnosable.

3.2.3. The Hypothesis Generation Issue

The main issue in hypothesis generation is in being able to generate the right hypotheses. The more accurately hypotheses can be generated the more efficient the solution may be. Alternatives available in hypothesis generation are rule-based, and model-based.

A rule-based approach to generating hypotheses will be based on the given symptom sets and produce a number of possible faults that account for the symptom sets. Problems may arise from incompleteness in the model as well as incompleteness in the rules.

A model-based approach to generating hypotheses attempts to reason from first principles about the artifact to determine what faults may have accounted for the symptoms. To do this the model of the artifact must be able to represent the artifact to the degree necessary to generate symptoms from faults. In one approach, possible faults are

hypothesized and injected into the model to generate symptoms. In the other, the model represents the dependencies between components of the artifact to a sufficient degree so that hypotheses may be "read off" of the model based on the given symptom sets.

Hypothesis generation is related to the symptom sets that can be generated from the model of the artifact. Thus, the effectiveness here is related to the symptom sets as well as the completeness of the rules in analyzing the symptom sets or the model in automatically computing the symptom sets for hypothesis generation.

3.2.4. The Single versus Multiple Fault Issue

In addition to the problem of mapping symptoms to faults there may be multiple dependent and independent faults giving rise to complex sets of symptoms. Fault diagnosis solutions must be able to determine when to generate multiple faults for a set of symptoms when the domain has that possibility.

In one approach, the set of symptoms is seen as a space to cover. As each fault is hypothesized, it covers some of the symptoms in the set. If there are remaining symptoms, additional faults may also be hypothesized [15]. In the approach mentioned above where faults are "read off" of the model, possible faults that account for the symptoms may include more than one actual fault; the minimal faults are always chosen first.

3.3. A Trade-off Matrix

Figure 7 illustrates trade-offs between the CPU and memory for the issues discussed above. The other major trade-off is effectiveness, however, effectiveness becomes more interesting in particular algorithms. In the figure we can see that using a quantitative approach to simulation will have a potentially large impact on CPU. We can also see that the memory requirements for quantitative versus qualitative simulation don't have any obvious trade-off. For symptom set generation, the static approach will need more memory to store the symptom sets while the dynamic approach will need to use CPU during run time to compute the symptom sets. For hypothesis generation the model-based approach may require more memory while CPU requirements are not an obvious advantage between rule-based and model-based.

		CPU	Memory
Simulator	Qualitative		
	Quantitative	√	
Symptom Set Generation	Static		√
	Dynamic	√	
Hypothesis Generation	Rule-Based		
	Model-Based		√

Figure 7

3.4. Example Systems

In this section some example systems for fault diagnosis are presented. The particular architecture chosen to specialize the fault diagnosis model has implications on the effectiveness of the solution. This includes whether the architecture handles multiple faults as well as single faults, whether it is a complete solution, i.e. can it be verified that every fault that can occur in the system can be represented and diagnosed with the architecture, and how easy it is to implement.

Three actual architectures that vary quite a bit will be discussed here. There are certainly more architectures in existence, but these three are representative of the variety of solutions possible. These architectures are: SSM/PMAD, The Patchwork Synthesis Algorithm, and deKleer's Diagnosing Multiple Faults algorithm.

3.4.1. SSM/PMAD

The SSM/PMAD system was developed by extensive front-end analysis of the power system being modeled [12]. The symptom sets for all the possible faults were computed statically by hand by placing faults at various places in the power system topology. Each fault would then cause the hardware to behave in a set way. This was

done before the hardware was actually built and required a power system engineer to help extensively with the knowledge engineering.

The function of noting discrepancies between expected behavior and actual behavior was performed by conventional algorithmic processes. A discrepancy was identified by noting that a switch tripped due to under voltage or over current for example. Other discrepancies could be identified by noting that Kirchoff's Current Law didn't hold for a node. As discrepancies are identified they are communicated to the fault diagnosis program (running on a Xerox 1186 LISP machine) which uses them to generate hypotheses.

It is at this point that the previously generated symptom sets come into use. The symptom sets were used to define a large number of rules to discriminate between possible faults. Then, when symptoms are identified and communicated to the fault diagnosis program, they are used by the rules to come to a hypothesis. For example, if the situation in Figure 2 occurred, the symptoms would be given to the fault diagnosis program which would be able to determine at least the two fault scenarios in both Figure 2 and Figure 3. At this point the fault diagnosis program would open all the affected RPCs. Then the 3k RPC that tripped on fast trip would be closed. If the 3k RPC trips on fast trip at this point we have the situation of Figure 2. If the 3k RPC does not trip when we close it, we go on and start closing the switches below it. When we get to the last switch in Figure 3 and repeat the fault situation (because we connect power to that short again), we know that we have a short at point B and that the 1k RPC above point B has a faulty sensor.

The fault diagnosis program for SSM/PMAD has proven to be very efficient. It only addresses single faults and therefore is not effective for multiple faults. Its effectiveness depends on how complete the analysis of the symptom sets was. The drawback we found with this approach was in its modifiability. It is very difficult to change the knowledge as the power system topology changes. This is more a symptom of the particular encoding of the knowledge in our system than rule-based systems in general. However, if we wanted to apply our system to other power system topologies that have different switch characteristics, we would have to generate a new symptom and fault matrix for that system.

3.4.2. The Patchwork Synthesis Algorithm

The Patchwork Synthesis algorithm addresses fault diagnosis for terrestrial power systems [15]. It makes use of algorithmic processes as well as a rule-based approach to hypothesizing faults which is based on OPS5. It also keeps a model of the power system

for generating symptom sets. This algorithm addresses both single and multiple faults in a heuristic fashion.

In Patchwork Synthesis, algorithmic processes manage the discrepancy identification problem. Discrepancies are then communicated to the fault diagnosis program in the form of symptoms. This is analogous to the SSM/PMAD approach. At this point the similarity ends. The Patchwork Synthesis approach uses rules that are triggered based on certain patterns in the symptoms. These rules then hypothesize various fault classes. Various faults, based on the fault classes and symptoms, are proposed to account for the symptoms. These are injected into the model of the power system to predict the symptoms that result. After collecting the sets of symptoms resulting from modelling the various faults, they are then used to *cover* the original symptoms of actual faults. Thus, a symptom set for a hypothesized fault can be subtracted from the original symptoms. If no symptoms remain that fault is hypothesized, otherwise other faults are also hypothesized. The goal is to get the best cover of the symptoms without much unnecessary overlapping and unreported symptoms.

This algorithm is more robust than SSM/PMAD. It handles multiple faults and unreported symptoms much more effectively. Obviously, more work has to go into developing the model so that symptoms may be generated correctly. This is in addition to the knowledge engineering needed to hypothesize faults based on original symptoms. Here we see one difference resulting from more work up front with slightly more CPU in fault discrimination resulting in a system that seems to be more effective. One difference though is that in SSM/PMAD, actual testing of the power system can be done while the Patchwork Synthesis algorithm does not do any testing except by simulation.

3.4.3. Diagnosing Multiple Faults

The architecture for diagnosing multiple faults [10] proposed by deKleer is based on the Assumption Based Truth Maintenance (ATMS) algorithm [9]. This algorithm diagnoses single and multiple faults, works with unreported symptoms, computes symptom sets dynamically (yet efficiently), uses a form of qualitative simulation and generates hypotheses using a model. The algorithm hinges on the ability to represent the model of the artifact in the ATMS. The possible faults for a symptom are then "read off" of the ATMS and combined to generate a set of minimal fault candidates.

To illustrate this method Figure 1 is reproduced as Figure 8 with annotations. The annotation on each switch is simply a symbol representing the assumption that the switch is

operating correctly. The cables are not represented as being faultable per se and represent the dependency of the lower switches on the upper switches.

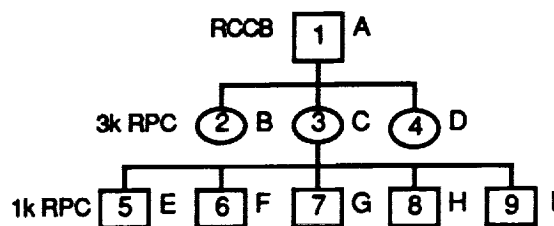


Figure 8

Assume that switches 7 and 9 trip. This represents a symptom set of two symptoms. Initially we look at switch 7. We see that if assumption G, C, or A was wrong, that we could account for switch 7 giving a symptom. For switch 9 our set is {I, C, A}. These assumptions are recorded at all the switches and are retrieved at no cost of CPU. The next step is to hypothesize the faults that could account for these symptoms. Initially we have an empty set of faults: []. We take the first set of assumptions for switch 7 and add each of them to all of the fault sets so far: [{G}, {C}, {A}]. Then we continue by adding the set of assumptions for each symptom to the fault sets. Additionally, we only keep the *minimal* fault sets. When we add the set of assumptions for switch 9 we get: [{G, I}, {C, I}, {A, I}, {G, C}, {C, C}, {A, C}, {G, A}, {C, A}, {A, A}]. This minimizes to: [{G, I}, {C}, {A}]. What this final set tells us is that if we had a fault and both assumptions G and I are wrong (that is, switches 7 and 9 are not operating correctly), then that would account for the observed symptoms. Or, if assumption C was wrong, that would account for the observed symptoms, etc. Obviously we would want to look at the minimal faults first.

In the above example where we ended up with final fault hypotheses of {G, I}, {C}, and {A} we might wonder if perhaps both assumptions {C} and {I} might be wrong. The above representation does account for this. Figure 9 shows the lattice representation being used to minimize the candidates. If testing shows us that the candidates alone do not account for the observed symptoms, we can move up the lattice to test more complex fault scenarios.

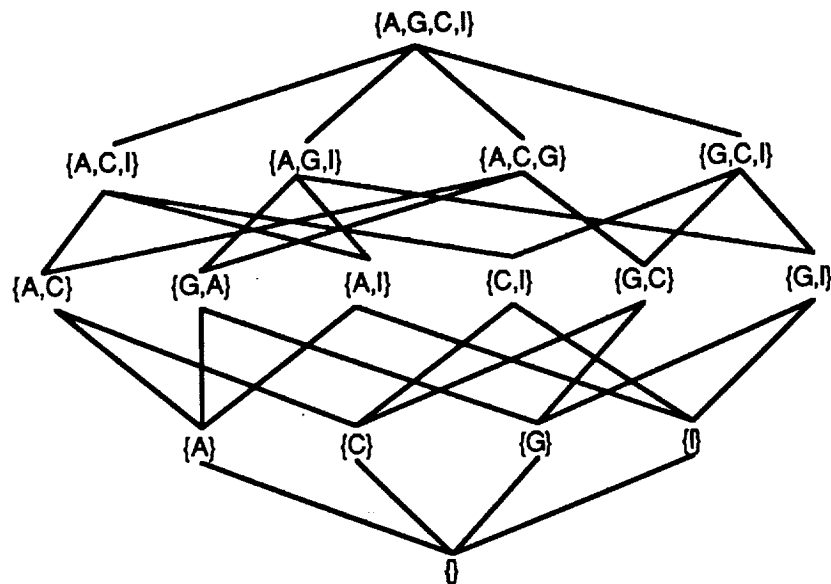


Figure 9

This algorithm is very efficient at run time. It has two drawbacks though. First, the dependency network representation may give rise to 2^n possible assumption sets where n is the number of components being represented. The second drawback is that of representation in general. It is very difficult to represent a complex domain (such as power system topologies where faults can occur in more than one way based on characteristics of switches, etc.) in the ATMS model. The example given looks good but it is really only saying that a switch either works or doesn't. In more realistic domains we may have to generate many assumptions for each switch: the fast trip sensor works, the I²t sensor works, the under voltage sensor works, etc. This is a very complex task and may end up requiring large amounts of space to store the necessary ATMS network.

4. Conclusion

The above examples as well as the discussion of the various issues have shown that there exist a wide variety of alternatives available when a particular fault diagnosis system is to be specialized from the general model. Depending upon the domain in question and the various constraints available (CPU, memory, knowledge engineering), a number of different solutions may be arrived at.

It is certainly possible for most of these solutions to work with multiple faults. That is more a property of the problem solver part of the model than the model and

simulator themselves. The trade-offs occur in amount of CPU available, efficiency required and the ability to represent the artifact in one model over that of another. The example of deKleer is one of the most interesting implementations, but probably one of the most difficult to apply to a domain. In almost any domain there will probably need to be some combination of rule-based processing using at least a qualitative model to generate expectations. Whether symptom sets are generated statically or dynamically may depend upon the constraints of the environment.

Acknowledgements

This work was performed by Martin Marietta Astronautics Group under contract number NAS8-36433 to NASA, MSFC, Huntsville, Al.

5. References

- [1] Adams, Thomas L., "Model-Based Reasoning for Automated Fault Diagnosis and Recovery Planning in Space Power Systems," IECEC 1986.
- [2] Davis, Randall, "Diagnostic Reasoning Based on Structure and Behavior," Artificial Intelligence 24, 1984.
- [3] Genesereth, Michael R., "The Use of Design Descriptions in Automated Diagnosis," Artificial Intelligence 24, 1984.
- [4] Genesereth, Michael R., "Diagnosis using Hierarchical Design Models," Proceedings of the National Conference on Artificial Intelligence, 1982.
- [5] Gilmore, John F., and Gingher, Kurt, "A Survey of Diagnostic Expert Systems," SPIE Vol. 786 Applications of Artificial Intelligence V (1987).
- [6] Hamscher, Walter, and Davis, Randall, "Issues in Model Based Troubleshooting," MIT Industrial Liaison Program Report 11-34-87, 1987.
- [7] Hester, Tom, "FIES II: A Real Time Fault Isolation Expert System," IECEC 1986.

[8] Iwasaki, Yumi, and Simon, Herbert A., "Causality in Device Behavior," Carnegie-Mellon University, Department of Computer Science, Technical Report CMU-CS-85-118, 1985.

[9] deKleer, Johan, "An Assumption Based TMS," Artificial Intelligence 28(2) 1986.

[10] deKleer, Johan, and Williams, Brian C., "Diagnosing Multiple Faults," Artificial Intelligence 32(1) 1987.

[11] Lee, S.C., and Lollar, Louis F., "Development of a Component Centered Fault Monitoring and Diagnosis Knowledge Based System for Space Power System," IECEC 1988.

[12] Miller, W., et al, "Space Station Automation of Common Module Power Management and Distribution," Martin Marietta Aerospace Denver Astronautics Group, 1989.

[13] Pearce, D.A., "The Induction of Fault Diagnosis Systems from Qualitative Models," Proceedings of the American Association of Artificial Intelligence, 1988.

[14] Reiter, R., "A Theory of Diagnosis From First Principles," Artificial Intelligence 32(1) 1987.

[15] Talukdar, Sarosh, and Cardozo, Eleri, "Patchwork Synthesis and Distributed Processing for Power System Diagnosis," IECEC, 1987.

INTELLIGENT SPACE POWER AUTOMATION

**A PAPER PRESENTED AT THE IEEE INTERNATIONAL SYMPOSIUM ON
INTELLIGENT CONTROL IN SEPTEMBER 1989.**

INTELLIGENT SPACE POWER AUTOMATION

Joel Riedesel
Chris Myers
Barry Ashworth
Martin Marietta Astronautics
Denver, Colorado 80201

ABSTRACT

Scheduling and monitoring satellite power systems has traditionally required intensive ground support. In general, ground support consists of managing the distribution of power to required loads, diagnosis of power system failures, and control of failure recovery. We at Martin Marietta, under contract since 1985 to NASA, Marshall Space Flight Center, have designed and implemented a power system test bed for Space Station Freedom modules that utilizes intelligent control in automating power system management and distribution as well as providing power system fault diagnosis and recovery.

Keywords: Power Automation, Fault Diagnosis, Scheduling, Intelligent Control

1. Nomenclature

AI	Artificial Intelligence
ADC	Analog to Digital Card
ECLSS	Environmental Control and Life Support System
FRAMES	Fault Recovery and Management Expert System
GC	Generic Controller
LC	Load Center
LLP	Lowest Level Processor
LPLMS	Load Priority List Management System
MSFC	(George C.) Marshall Space Flight Center
NASA	National Aeronautics and Space Administration
PDCU	Power Distribution Control Unit
SIC	Switch Interface Controller
SSM/PMAD	Space Station Module (Autonomous) Power Management and Distribution

2. Introduction

This paper describes a power system testbed, Space Station Module Power Management And Distribution (SSM/PMAD), that was designed for intelligent control. This testbed was designed from the ground up, the hardware (both computer controlling hardware as well as switches and controller cards) and software was designed with the goal of an autonomous system in mind. The focus of this paper is on the intelligent control of the power system. In that respect we do not discuss the design of the physical hardware although that is very important to intelligent, autonomous control. The design for intelligent control is discussed in the remainder of this paper while we acknowledge the important role of the physical hardware without going into any detail about it.

The test bed we have developed may be described in a number of levels. At the bottom level exists the switchgear hardware while at the top level scheduling and fault diagnosis resides. Figure 1 shows the topology of the power system and where the higher level functions fit in¹. In between these levels exists a variety of software and hardware designed to further enable the intelligent automation process which supports the higher processes. We will describe the supporting hardware and software of the breadboard in the context of its global goal of intelligent power automation. Sections two, three, and four will then talk

¹ On December 14, 1988 a NASA Change Request specifying 120 Vold dc source power was put into effect. Also, since then the bus topology has changed to a STAR.

about how intelligence is distributed among the various software components of the system in detail. An important aspect of the system approach to space systems power management described here is that 1) intelligent control is combined with 2) fault management and 3) activity & resource planning and scheduling to form the nucleus of a power management and distribution system manager. Also, deterministic processes which may, by definition, belong to knowledge based operations have been removed and distributed to more fundamental lower level activities, wherever possible.

2.1. Problem Definition

The goals of reducing ground support costs and increasing feasibility of long term long distance missions motivates the intelligent automation of space power management. The Space Station Freedom modules require the management of power to 44 distribution points each with an average of a dozen loads – larger than anything yet flown.

The automation of power management and distribution involves three primary subtasks: Scheduling of power to particular switches for operation of loads, control of the power distribution hardware during schedule execution, and failure diagnosis and recovery during fault occurrences.

2.1.1. Scheduling The scheduling problem can be an intensive task. Witness the example of SKYLAB. Mission operations had to be scheduled in real time by a large ground support task force. The result was 90 man-months of labor to schedule 9 months of tasks. 10 man-months per mission month. Intelligent automation of scheduling, especially in light of changing power availability and larger power systems, becomes increasingly necessary (it is estimated 50 or more ground personnell will be required for Space Station Freedom).

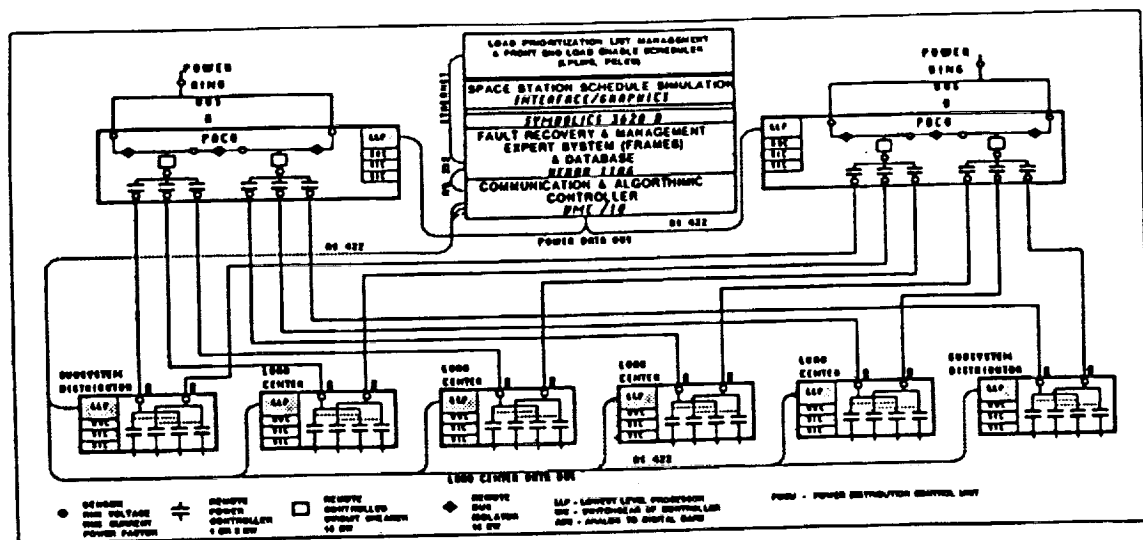
In general, the scheduling problem is an NP-complete problem (the number of solutions may be exponential) requiring the skilled use of heuristics to manage the problem. Scheduling asks to optimize a set of tasks with respect to efficiency. If every possible solution were to be analyzed the problem might never be solved.

In the domain of satellite power systems the scheduling problem requires a large amount of knowledge about the loads being scheduled. Loads have diverse and changing energy requirements. Loads may also be constrained to operate in determined periods of time based on, for example, the visibility of certain star clusters, the sensitivity of the load itself (e.g. does it require a minimal amount of vibration to operate correctly), and even on the availability of crew members to operate the loads in a required manner. Loads may also have priorities, life support systems being one of the most critical.

2.1.2. Control of the Power Distribution Hardware Control of the power distribution hardware is done by the Lowest Level Processors (LLPs). Each LLP is responsible for commanding switches open and closed in response to scheduled activities, for maintaining power to loads in as efficient a manner as possible, and for buffering data between the switches and the higher level processes.

The LLPs are constantly aware of the amount of power being consumed by the switches and automatically turn switches off if the amount of power being used is more than allowed (as indicated by the schedule). The LLPs are also aware of the priorities between loads at a single load center. If the amount of power available is suddenly reduced, the LLP can continue to power those loads that have higher priorities and drop those loads with lower priorities.

The LLPs are also used to enable the Fault Recovery and Management Expert System (FRAMES) to gather information from the power system. During fault diagnosis, FRAMES may send switch commands to the LLPs which are immediately executed and the resulting information sent back. The LLPs contain all of the system low level activities which serve to interface the knowledge based intelligent control activities to the power distribution hardware.



2.1.3. Failure Diagnosis and Recovery In the event of a failure in the power system, automation of the diagnosis of the failure is required if automation of scheduling is desired. The process of diagnosing a failure in the power system and dynamic scheduling around the failed parts of the system defines a mode of failure recovery.

The primary motivation of automated failure recovery is to continue operation with minimal interruption of the loads. By scheduling around the affected parts of the power system it becomes possible to manually repair the affected regions of the power system for subsequent use, without shutting down the entire system.

There may be many different faults occurring in a power system. These include a single fault detected by a switch tripping, usually indicating an open circuit or a short circuit. If a switch is not operating correctly a masked fault may occur where the switch does not trip but the one above it (closer to the source) does. Fault diagnosis is also complicated by multiple independent faults occurring in the same time period. The diagnosis of faults depends to some degree on both the placement and accuracy of sensors for fault detection as well as the ability to manipulate switches in the power system in an attempt to isolate the location of the fault and provide confirming evidence of a particular fault.

Intelligent manipulation of switches in the power system domain as a fault isolation technique is dependent on the hierarchical nature of the switches. This is possible due to the acyclic nature of power systems (where power flows from the source to the loads, guided by switches).

2.2. Closed-Loop Control

2.2. Closed-Loop Control

The system architecture yields a nested closed-loop control system approach. This is shown in Figure 2. The outer control loop, operating in seconds of time, employs activities which are managed by the knowledge-based control functions, while the inner control loop, operating in microseconds to milliseconds, utilizes conventional limit checking techniques. The result is that situations which require immediate response are managed within the inner loop where system safe-haven can immediately be reached and the schedules are considered fixed. But, situations which can allow more processing consideration are explored and managed at the outer control loop, providing control inward to the Lowest Level Processors (LLPs). Future parameters, such as updated schedules and updated priority lists, are produced outside of the control activity performance constraints to prevent operational conflicts.

2.3. The Power System

Our current implementation contains eight Lowest Level Processor (LLP) units. Every LLP contains a computer controlling two Switchgear Interface Cards (SICs). Each SIC card is capable of interfacing the computer to an Analog to Digital Card (ADC) and up to fourteen Generic Controller (GC) cards, which control the switches. Each LLP may control as many as 28 switches.

From the perspective of power system automation there are two features of the physical power system to be noticed. The first feature is the architecture; the power system is laid out as a distributed system. Each LLP is controlling a set of switches. If one LLP fails the other LLPs are not affected (in general). In the current implementation there are two LLPs that control distribution of power to the other six LLPs. This gives us a hierarchy of switches. The second feature is communication; the LLPs are used for

communicating both scheduled switch commands (open and close) to the switches as well as switch and sensor information from hardware back to FRAMES.

The LLPs exist within two environments; the Power Distribution Control Unit (PDCU) and the Load Center (LC) (as shown in Figure 1). For both environments they carry out numerous activities. However, functionality is not completely uniform between the two distribution controllers. For both, the scheduled operations of switching activities are carried out. They both acquire and process switch and sensor data in the form of current and voltage, and pass that data up to FRAMES. They also perform range testing and fault testing on the data, providing notification to FRAMES in the event anything out of limits or faulted is found. Both environments also calculate and pass system performance statistics to the higher level processes. The main difference between them lies in how limit exception conditions are handled. In the LC, if a load draws too much current the load is shed. In a PDCU this is not done, instead the PDCU notices the out of limit condition and informs the fault recovery and diagnosis system of the exception. This may then lead to recognition of a soft fault or a problem with one of the loads below the PDCU.

2.4. Scheduling

Scheduling the activities needing power is the first step in the operation of power management and distribution. Scheduling is initiated by the user and subsequently performed whenever there is an unforeseen change in the power system. The goal of the scheduling process is to make use of the available power in the most efficient manner possible, taking into consideration load dynamics and resource availability, among other things.

When a fault occurs during operation, FRAMES diagnoses it and lets the scheduler know which switches can no longer be scheduled. Rescheduling resumes as many activities as possible and isolates the faulted areas until they may be repaired. Rescheduling will also take place if the user specifies that there will be a change in the available power to the system for some future period of time.

2.5. Fault Recovery and Management

The purpose of the Fault Recovery and Management Expert System (FRAMES) is to keep the power system operating as completely and effectively as possible, especially in failure situations. This is done by using a model of the power system network, keeping track of both what is scheduled to happen and what is actually happening in the power system to recognize any discrepancies in its operation. When a discrepancy is detected FRAMES analyzes the situation to determine what fault will account for the discrepancy.

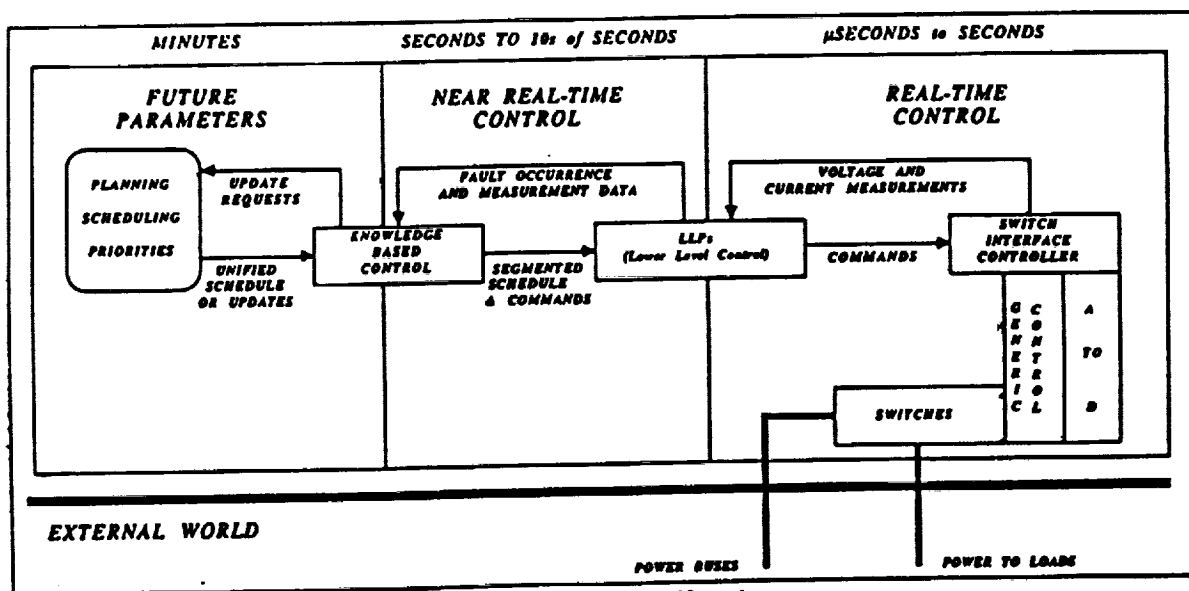


Figure 2 - Automation System Control Interactions

The LLPs communicate various data about their switches and sensors to FRAMES. This includes current data, switch status information, performance data, and trip related information. When FRAMES discovers that there are symptoms indicating a failure in the power system, analysis determines what class of faults may account for the symptoms, and further testing may isolate the fault to a particular location in the power network. Once the fault location has been isolated as much as is possible (given available information from sensors and switches), FRAMES determines and communicates to the scheduler those switches that are no longer useable. The scheduler uses this information to schedule loads on the remaining usable switches.

FRAMES is designed to diagnose faults occurring both independently and at different times from one another. This is in contrast to some forms of multiple fault diagnosis. The types of faults FRAMES is designed to diagnose include hard faults, soft faults, incipient faults, cascaded faults, and masked faults. Hard faults are defined as faults that cause switches to physically trip. Soft faults are illegal uses of current in the power system, such as resistive shorts to ground. Incipient faults are situations that will become hard faults in a reasonably short time if nothing is done to avert them. Cascaded faults are faults that result in a larger number of symptoms than the actual fault requires, for example, if a switch in the middle of a hierarchy of switches trips, all the switches that were operating below it may also trip on under voltage. Masked faults are faults that can only be detected in the presence of other faults, they may be the result of failed or inaccurate sensors for example.

2.6. System Architecture

Figure 3 depicts the logical information flow in the system to support intelligent automation. Various aspects of this picture will be described in the following three sections explaining the data needed to support intelligent automation.

3. Intelligent Control by Scheduling and Dynamic Contingency Management

The scheduling of resources and activities is an important autonomous level of intelligent control for space power systems. This knowledge based activity includes substantial knowledge of the power system itself as well as knowledge of the activities requested by the user. Actual loads are only roughly characterized. The scheduling problem consists of creating a schedule for the requested activities making the most efficient use of resources as possible. The problem is further complicated by a changing environment causing possible problems or power losses in the power system. Scheduling must also be dynamic in order to maintain efficiency and autonomy.

The basic power system resource is power and the ability to enable power properly. Other resources may include crew, tools, and switches.

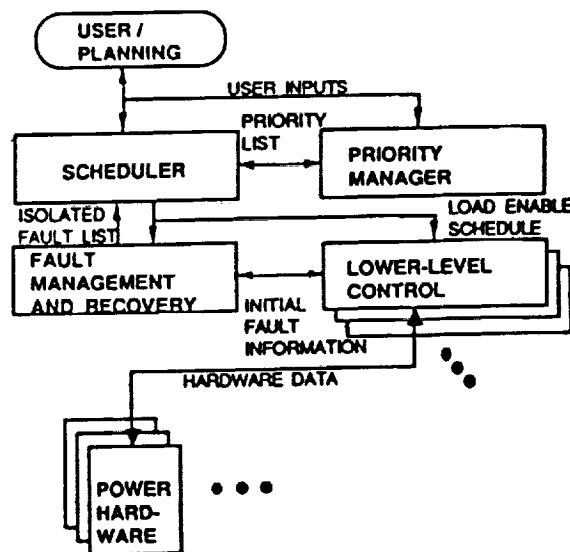


Figure 3 - Logical Information Flow

The power system topology and switch characteristics actually complicate the availability of power. Power is constrained to certain levels at the various load connection points based on switch characteristics and connections.

The rest of this section will discuss first activities and constraints, important concepts to the scheduling problem, followed by schedule generation and dynamic rescheduling.

3.1 Activities

Activities are represented hierarchically. An activity group is a set of activities representing different ways to accomplish a particular goal. An activity in turn, is a linear sequence of subtasks which, when performed in the order specified, satisfy the goal. A subtask is a portion of an activity whose resources and condition requirements do not vary over its duration. Duration can vary as can delays between subtasks. Finally, activities are also annotated with a priority rating, a subjective scale of the importance of the activity. For example, on Space Station Freedom, ECLSS (Environment Control Life Support System) will probably have a very high rating while astronauts are present.

3.2. Constraints

Constraints represent conditions on or between activities. They arise for a variety of reasons. Resource types and availability give rise to rate-controlled and consumable resources. Certain requirements that need to be met for proper operation of a task define conditions on activities. Opportunity window constraints, representationally equivalent to conditions, are constraints not associated with a resource but necessary for the performance of an activity. These constraints, rate-controlled and consumable resources, conditions, and opportunity windows are all performance-controlling constraints.

Rate-controlled resources are those whose availability continues when the subtask using them ends. Examples of this are crews time, thermal rejection, electrical power and equipment. Consumable resources, on the other hand, once depleted, stay depleted until some activity specifically replenishes them. Water, liquid nitrogen and lubricating fluids are examples of this type of constraining resource.

Conditions are states the spacecraft must maintain in order to perform a subtask, and include spacecraft attitude and position, temperature ranges, acceleration, vibration, etc. In general, conditions cannot be consumed by an activity requiring them, which differentiates them from rate-controlled resources. An opportunity window is a performance-controlling constraint not associated with the availability of any resource, but constraining the performance of a subtask just like a constraint would. Activities with opportunity window constraints must have appropriate subtasks scheduled to happen within them.

Many of the performance-controlling constraints can be satisfied by more than one resource or condition. An example of this is the case where a subtask could be performed by either of two crew members trained to use a particular piece of equipment, but not by any of the other crew members. This is referred to as a resource disjunction, a case where one resource or another can satisfy a requirement. The existence of a resource disjunction in a subtask description greatly increases the difficulty of finding times during which a subtask can run, as opportunities to perform the subtask depend on which resource is chosen - leading to exponentially large search spaces. This can be further complicated by the fact that a resource choice in one subtask can control that in another, e.g. the crew member who performs the calibration of an instrument should be the same one who read the manual at the start of the activity.

Another basic type of constraint on activities is the relational constraint. Constraints of this type relate the start or end of one subtask to that of another, either in the same activity or in another. A relational constraint may also constrain activities by relating the start or end of a subtask to some event or absolute time on the timeline.

3.3. Schedule Generation

A schedule is created by repeatedly executing three steps, referred to as the select-place-update cycle. The first step involves evaluating every activity requested for scheduling with respect to a set of selection criteria, and choosing one activity to put on the schedule next. These criteria include the base priority associated with each activity, the percentage of performances requested that have been scheduled for each (success level), and the relative constraint of each (opportunity). Relative constraint is a rough measure of how many different opportunities each activity has to be placed on the schedule. These criteria are combined using user-selectable weights which reflect the importance of each criterion to the user. An activity chosen will have higher priority, a lower percentage of requested performances scheduled, and/or fewer opportunities to be scheduled than other activities.

Once an activity has been chosen to be scheduled, one instance of it is placed on the schedule. The calculation resulting in the measure of constraint actually determines all allowable start and end times of all

subtasks in each activity. This information can be used during placement to position the performance according to soft constraints (preferences) imposed by a user. The user, for example, may maximize the data collection subtask, or can schedule the activity as early or as late in the scheduling period as possible. If there is a resource disjunction in a subtasks's requirements, a preference can be specified and adhered to, and in fact, a set of possibly contradictory soft constraints can be specified along with an ordering of their importance.

The final step in the scheduling cycle involves updating resource availability profiles to reflect the activity's consumption of resources. The cycle then repeats for as long as the user wishes or until there are no opportunities to schedule any activity. The combination of weights on selection criteria and attention to soft constraints during placement allows the scheduler to be tuned for a variety of scenarios.

3.4. Dynamic Rescheduling

There are a number of situations in which a schedule must be altered in other ways to accommodate various changes. It may become known that resource or condition availabilities will change or have changed, or that an activity not previously known about needs to be added to the schedule. These situations are handled within the scheduler by a heuristically-guided unscheduling mechanism in concert with a method of altering descriptions of activities already in progress, and aided by the maintenance during scheduling of multiple partial schedules. A change in resource availabilities may result in a projected over-use of a resource. When a resource is found to be overbooked, all activities using that resource during the time it is overbooked are evaluated. The evaluations are done according to a set of criteria designed to determine what activity to alter or unschedule to solve the problem. This should be done with the least impact on the schedule. The criteria include how well the activity's use of the resource fits the amount of overbooking, whether the activity is in progress or not, the activity's priority, amount of crew involvement, use of other resources, further opportunities to be scheduled, success level, and others. These criteria are also weighted to allow flexibility to a user. An activity is selected and unscheduled or selected and altered, then all are again evaluated and another unscheduled or altered, until no resource overbooking remains.

Activities whose condition constraints are violated must be altered or unscheduled; there is no choice as to which to affect. These are all handled the same as those chosen to be perturbed by a resource overbooking. When it is determined that an activity not scheduled must be added to the timeline, the scheduler first tries to find a way to schedule it which will not disturb anything already scheduled. If no opportunity exists, the scheduler will try to find opportunities which will result in only lower-priority activities being perturbed, and if found, will unschedule or alter one or more of those using the same techniques as overbookings. If no lower-priority activities can be found to bump, the scheduler rejects the request (perhaps the activity is not schedulable even in the absence of other activities, or all interfering activities are of higher priority).

The last thing the scheduler tries to do after altering the schedule in a contingency is to schedule any activities whose requests have not been fully met, possibly using resources released when some other activity was altered or unscheduled.

3.5. Operation

In the operation of the scheduler with respect to the rest of the automation system, the scheduler sends the current (or contingency) schedule to both the fault diagnosis system and the lower level intelligent processes. In addition to the schedule, a priority list is computed by a Load Priority List Management System (LPLMS) and sent down on a regular basis for the lower level processes to use in the event of load shedding situations.

4. Intelligent Control by Fault Management and Diagnosis

The fault management and diagnostic system provides support for the autonomous power system by intelligently monitoring faults and intelligently controlling the power system for fault isolation. Autonomous operation of the power system would not be possible without a fault recognition and isolation knowledge based activity to support dynamic rescheduling and overall control of the power system.

The general fault diagnosis problem consists of three parts: Fault detection, fault isolation, and fault recovery. Fault detection is taken care of by the lower level processes in general. For some faults (e.g. some soft faults, and incipient faults) fault detection must occur at a level above the lower level processes. Fault recovery is enhanced by the scheduling system, with FRAMES which also carries a global picture of the task to be accomplished.

There are a number of issues involved in fault diagnosis in general (see [9] for a general overview). These include: The computation of symptom sets, model based reasoning, single vs. multiple faults, and how fault isolation is done. Each of these issues will be discussed here.

4.1. Symptom Sets

A symptom set is a set of symptoms that indicates a fault. To put it another way, a fault gives rise to a set of symptoms. In fault diagnosis, the symptom sets may be computed in a number of ways. One may have a model of the power system dynamically compute the possible symptom sets for any possible fault in the power system. Alternatively, one may analyze all the possible faults in the power system beforehand and save the dynamic computation by using memory space instead. The benefit of dynamic computation is to be able to compute symptom sets for unforeseen power system topologies. In the static computation mode, if the power system topology changes significantly, potentially large amounts of work may need to be redone.

Symptom sets are used for pattern matching in an attempt to determine what fault may have occurred in the power system. A symptom or set of symptoms resulting from an actual fault may indicate more than one possible fault. It is then important to isolate the fault from among the various possibilities.

In FRAMES, all the reasonable fault scenarios are analyzed beforehand and their symptom sets computed. These are then used within FRAMES to pattern match against to determine possible fault situations.

4.2. Model Based Reasoning

Similar trade-offs apply here as in the computation of symptom sets. In general, the motivation for using model based reasoning is when all the fault scenarios are not necessarily knowable beforehand. This usually happens when requirements are changing or when the domain of reasoning is a dynamic domain. Model based reasoning may also be used when reasoning from first principles is required [8].

For FRAMES, a model is used to help in the gathering of symptoms from the LLPs. As all possible fault situations were analyzed beforehand, it was not needed for symptom set computation. When symptoms are detected at the lower level processors, they will be sent to FRAMES. As the lower level processors implement a distributed system, FRAMES needs to be able to compute, based on given symptoms so far, what other symptoms other lower level processes may yet provide. Thus, the model is used for generating symptom set expectations or profiles.

4.3. Single vs. Multiple Faults

The issue of single vs. multiple faults is whether FRAMES will diagnose those faults that occur singly, spaced out from one another, or if FRAMES will diagnose independent or dependent faults occurring at or near the same time. Diagnosing multiple faults is not a simple issue. The dependent nature among faults greatly increases the complexity of the situation. Furthermore, multiple simultaneous faults were not considered very credible scenarios for the domain under consideration. For FRAMES, single fault diagnosis is utilized. FRAMES also diagnoses certain classes of multiple faults – the masked faults. For example, if a switch's current sensor is broken and a short appears below the switch, the switch above will trip on over current. FRAMES will diagnose these kinds of faults.

There is another type of fault situation, cascaded faults, that applies to both multiple faults and single faults. A cascaded fault situation is where a short circuit may arise below a 3k switch causing it to trip on fast trip. Consequently all the switches below it that were closed will also trip on under voltage. This is a cascade effect. To be accurate, what is really being reported to FRAMES is a set of cascaded symptoms arising from a single fault in this example. Most faults will have cascaded symptoms giving some further indication of the fault. Thus, when FRAMES diagnoses faults, it also includes those faults with cascaded symptoms. This phenomenon forces FRAMES to explore multiple fault scenarios for single fault occurrences.

4.4. Fault Isolation

The issue of fault isolation is how to isolate where a fault occurred. Symptoms may describe a large class of possible faults that could account for them. Obviously, one does not want to hypothesize all the possible faults. Rather, one would like to discriminate further between the possible faults. There are two basic mechanisms to do this. One is to dynamically probe for values at various points in the power network. Obviously, in a fully automated system this is not easy to accomplish. The second basic method is to manipulate the switches.

Switch manipulation is performed in FRAMES and provides for control of the state of faulted areas of the network, allowing testing by opening and closing switches to produce useful results. As switches are opened and closed, data are collected from the results of these operations to further discriminate between possible faults. Switch manipulation proves to be a very useful diagnostic tool in power networks due to the hierarchical topology of the switches, as in Space Station Freedom for example.

4.5. Operation

During normal autonomous operations, FRAMES performs a monitoring function. When a fault occurs, the lower level processes send symptom set data to FRAMES. FRAMES does pattern matching over the received symptom set and determines if further testing must occur to isolate the fault. If so, switches are manipulated to generate further discriminating symptoms. Once the fault is isolated, the switches that can no longer be operated due to the fault are communicated as being out of service for the rest of the scheduling period to the scheduler. FRAMES then resumes normal monitoring operations.

5. Intelligent Control by Smart Low Level Functions

Physical control of switches is achieved by intelligently commanding switches open and closed, by considering the power scheduled to be used through the switches, and by managing priorities of loads using switches at the LLPs. Characteristics of this intelligent control include: Distributed control, load shedding, redundancy management, scheduled operations, and algorithmic control of loads and switches. This is actually a deterministic extension of FRAMES.

The low level functions provide intelligent control of the switches by considering power scheduled and priority of loading. The architecture of the system dictates that the low level functions be resident at every LLP yielding distributed control of the switchgear. The loading and power usage is controlled algorithmically through command of the switchgear. The low level functions are also responsible for preliminary fault detection. Knowledge supplied by the scheduler and the priority manager is used in conjunction with the data received from the switchgear to intelligently control switching operations and the distribution of power.

One of the features of the low level functions is distributed processing and control. Because each LLP controls a limited number of switches, and there are many LLPs throughout the system, control of switches is distributed at the LLP level. Since the low level functions operate concurrently on each LLP, the overall system has a faster response to fault situations. Also, the loss of a single LLP will not be catastrophic to the power system. Performance information is maintained at each LLP where processing power is least expensive with respect to system performance.

Although distributed processing is advantageous for processing and survivability, it also has some potential disadvantages. Distributed control of the system has consequences in global operations. The most important consequence of distributed control becomes apparent in global load shedding. When the system is required to immediately reduce the amount of power consumption due to change in source availability, the LLPs are required to bring their power consumption into line with availability. The consequence is that a load of higher priority on one LLP may be shed, while a load of lower priority controlled by a different LLP remains powered. In operating independently of each other, the LLPs do not exchange information. Furthermore, an LLP has no idea what is loaded on any other LLP. Therefore, global load shedding cannot be easily accomplished at a local level, forcing resolution of the problem upward. Other global operations, such as planning, have similar problems which tend to force solutions up into central entities. The overall problem is solved by producing items with global implications at the system management level, and then performing component level management on those items at the LLPs.

Algorithmic control of the switchgear at the LLP is the primary activity of the low level functions. Control and monitoring of the switchgear can be broken down into several operations. First, accepting schedules and commanding the opening and closing of switches. Second, shedding switches by load priority, when necessary. Last, managing redundantly sourced loads. With these operations, algorithmic control begins to take form.

The acceptance and implementation of scheduled operations is fundamental to distributed management at the LLPs. A schedule is passed to an LLP and stored. When the effective time of the schedule is reached, the schedule is placed in operation. The events within the schedule are processed as individual events when their effective times are reached. There are only three types of events, turning on a switch, turning off a switch, and changing a switch's allowed power consumption or redundancy state. Each event contains limits on power consumption, redundancy and permission to switch to redundant, the switch identifier, and the time of an event. The algorithmic control only stores two schedules, the one presently in operation and the next schedule. If a contingency schedule is received, it is treated as the next schedule.

During periods of intelligent control, scheduling is the only way a switch may be turned on or off. All other switching activities must occur from a manual level of intervention where no automation functions are utilized.

Two other methods of turning off switches exist under algorithmic control. Both of these methods are categorized as load shedding. The first case is when a switch is scheduled on and a load of lower priority must be shed in order to supply sufficient power to enable the scheduled load. The process for shedding loads is priority based. The loads to be shed are marked in order of increasing priority until the amount of power required is reached. The shedding algorithm then proceeds through the marked loads in order of decreasing priority to see if any need not be shed while still meeting the power requirement for shedding. This process removes only those loads which must be removed. The other way a switch may be shed is if a redundant switching operation is scheduled.

Redundancy management is another feature of algorithmic control. When a load is scheduled at a load center, it can be both redundant and have permission to switch to redundant. If such a load trips, the switch powering the load from the redundant bus is scheduled immediately. Scheduling this load on the redundant bus could cause load shedding on that bus in order to accommodate the new load. This is a function of load priorities and available power and is the method that the LLP uses to deal with redundantly sourced loads.

The last function of the LLP to be discussed is fault isolation. When the LLP detects a hard fault at a switch or set of switches, it sends the fault type and switch information to FRAMES. FRAMES may request the LLP(s) to manipulate their switches so that it may accumulate knowledge about the fault. The basis of manipulating switches in a specific order is to get information about the fault from a known state of the system. Only by opening, flipping, and closing the switches at the LLP level can FRAMES reasonably attain more knowledge about the fault and its causes. Upon completion of its fault isolation, FRAMES informs the user and takes corrective action. This corrective action may be to take a particular switch out of service, and this will be reflected in the contingency schedule issued during fault recovery. Fault isolation is a very powerful tool for FRAMES and its data gathering is a low level function.

6. Acknowledgements

This work was performed while under contract to NASA, Marshall Space Flight Center, contract number: NAS8-36433.

7. References

- [1] Ashworth, Barry R., "An Architecture For Automated Fault Diagnosis," Proceedings of the Intersociety Energy Conversion Engineering Conference, 1989.
- [2] Britt, Daniel L., John R. Gohring, and Amy L. Geoffroy, "The Impact on the Utility Power System Concept on Spacecraft Activity Scheduling," Intersociety Energy Conversion Engineering Conference, 1988.
- [3] Freeman, Kenneth A., Rick Walsh, and David J. Weeks, "Concurrent Development of Fault Management Hardware and Software in the SSM/PMAD," Intersociety Energy Conversion Engineering Conference, 1988.
- [4] Geoffroy, Amy L., Daniel L. Britt, Ellen A. Bailey, and John Gohring, "Power and Resource Management Scheduling for Scientific Space Platform Applications," Intersociety Energy Conversion Engineering Conference, 1987.
- [5] Lee, S.C., and Louis F. Lollar, "Development of a Component Centered Fault Monitoring and Diagnosis Knowledge Based System for Space Power System," Intersociety Energy Conversion Engineering Conference, 1988.
- [6] Miller, W., E. Jones, B. Ashworth, J. Riedesel, C. Myers, K. Freeman, D. Steele, R. Palmer, R. Walsh, J. Gohring, D. Pottruff, J. Tietz, D. Britt, "Space Station Automation of Common Module Power Management and Distribution," Interim Final Report no. MCR-89-516, Martin Marietta, 1989.
- [7] Miller, William D., and Ellen F. Jones, "Automated Power Management within a Space Station Module," Intersociety Energy Conversion Engineering Conference, 1988.
- [8] Reiter, R., "A Theory of Diagnosis from First Principles," Artificial Intelligence 32(1):57-96, 1987.
- [9] Riedesel, Joel D., "A Survey of Fault Diagnosis Technology," Proceedings of the Intersociety Energy Conversion Engineering Conference, 1989.
- [10] Weeks, D., "Autonomously Managed High Power Systems," Intersociety Energy Conversion Engineering Conference, 1988.
- [11] Weeks, D., "Artificial Intelligence Approaches in Space Power Systems Automation at Marshall Space Flight Center," in the Proceedings of The First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Tullahoma, TN. 1988.
- [12] Weeks, D., "Automation of the Space Station Core Module Power Management and Distribution System," in the Proceedings of SOAR '88, Dayton, OH. 1988.

**AN OBJECT ORIENTED MODEL
FOR EXPERT SYSTEM SHELL DESIGN**

**A PAPER PRESENTED AT THE IEEE INTERNATIONAL CONFERENCE ON
COMPUTERS AND COMMUNICATIONS IN MARCH 1990.**

An Object Oriented Model for Expert System Shell Design

Joel D. Riedesel
Martin Marietta Astronautics Group
P.O. Box 179, MS: S-0550
Denver, Co. 80201

Abstract

In this paper a tuple space based object oriented model for knowledge representation and interpretation is presented. This model is currently being applied to the expert system arena of knowledge base systems. The architecture addresses two important issues of knowledge base system design. One, the trade-off between expressivity and tractability in knowledge representation and two, knowledge base system and expert system shell domain independence.

Current expert system shells stress how many rules can be fired per second. The most common of these is OPS5 using the RETE pattern matching algorithm. The speed of expert systems is very important, but at the same time it is important to be able to represent domain knowledge adequately. This is where the expressivity of a knowledge representation language becomes important. For complete expressivity one might turn to first order logic at the expense of potential intractability. An architecture is defined here that falls somewhere between OPS5 and FOL in expressivity and tractability. It trades some expressivity for tractability and vice versa.

Most expert system shells today also tend to be somewhat domain restrictive (such as diagnostic and classification domains). Both the knowledge representation language and the inference strategy used contribute to this restrictiveness. Analysis of requirements for greater expressivity of both knowledge and inference strategy for representing domain independence as well as for keeping intractability manageable has resulted in an object oriented language for defining knowledge base management systems as well as their instantiation. This language and its implementation will be discussed in some detail.

1 Introduction

Early Artificial Intelligence (AI) programs centered around general reasoning mechanisms. This included sound and complete search methods as well as powerful knowledge representation languages (for example, General Problem Solver [15] and resolution theorem proving [18]). These early efforts worked well on small problems but failed to scale to large problem spaces adequately. In fact, NP-complete problems seemed to abound everywhere. As a result of the problem of intractability researchers moved toward domain dependent reasoning. Here, the knowledge could be restricted to domain dependent heuristics instead of general search, as well as restricted knowledge representation languages. These domain dependent expert systems became quite successful (see [2] and [19] for an overview).

Although these domain dependent expert systems worked quite well, expert system technology and knowledge representation of various expert system shells did not transfer well from one domain to the next. Witness the diversity of expert system shells currently existing in the commercial marketplace. The number of shells residing in the university research labs are probably an order of magnitude larger. This diversity is a result of two main restrictions (for combating the tractability problem). One, the restriction on knowledge that can be described or stated, the declarative knowledge. Two, the restriction on mechanisms of inference over the knowledge. Because of these restrictions, it became very important to analyze a domain very carefully to be certain that it could be represented in an existing shell.

The moral is that although generality is needed for a tool to be applicable to a large number of domains providing transfer of both knowledge representation and inferencing techniques, to beat the tractability problem domain dependent tools must be developed and applied. The thesis of this paper is that we can beat this moral. A knowledge representation language that has sufficient heuristic adequacy [21] can be both general and tractable. But the tractability provided is only because knowledge of the domain is described and developed using *software engineering* techniques. If general search strategies are continually relied on, instead of developing efficient heuristics applicable to the domain, tractability will still be an important issue.

1.1 An Overview of the Language

A general knowledge representation language and basic inference mechanism is described here.¹ This language is used for defining expert system shells and instantiating them to a specific domain. The language is defined and implemented using the Common Lisp Object System (CLOS) up to the level of rules. Rules, rule groups, and knowledge bases are also objects but of a different type. The language objects from rules² on up are defined in a user accessible object language using a frame system. The object oriented view provided at the higher levels allows the user to modify and extend the language very easily to make it applicable to a previously difficult domain.

The language (let's call it a Rule Management System (RMS) language) is supported by a tuple space [4] implementation of a database as well as a frame system built upon the database. The language provides tractability in two primary ways. One, tractability is supported by organizing knowledge in logical modules (taking advantage of divide and conquer methodology) and by providing heuristic control over knowledge. This is the software engineering approach to tractability. Two, tractability is further enhanced by providing a level of efficiency in the matching mechanisms for determining what knowledge is true in the world.

The RMS language may be instantiated in many different forms including more classical expert system shells. Figure 1 shows a general Rule Management System architecture. This architecture consists of three main parts, the database, database interface, and the rule management system. The database is independent of the rest of the RMS. It may be distributed or not. The database interface

¹ This language is part of a much larger system, KNOMAD — **K**nowledge **M**anagement **D**esign System, being developed by our group at Martin Marietta Astronautics. The Rule Management System module of it is the focus of this paper.

² Rules are represented as both a CLOS object and a user specifiable object. One could make all the objects of the language user accessible, but efficiency would be hampered.

maintains links between the database and the rules as well as providing abstraction over data in the form of frames. Finally, the RMS consists of a number of rule groups and mechanisms for performing inference over rules. The RMS and database interface are collocated in one location. Of course, many of these RMS and database interface modules may exist in different locations providing support for distributed knowledge agents as well.

2 Expressivity and Tractability

Expressivity has at least two meanings that may be distinguished. One meaning of expressiveness asks whether a language allows something to be said. In this meaning various languages can be compared in terms of formal language theory and classified as Type 0, 1, 2, or 3 [12]. We find that most computer languages are Turing equivalent and therefore it becomes moot to compare languages on the simple basis of whether or not something can be said in them. The other meaning of expressiveness asks how easy can it be said, how clear or understandable is it using the language? This meaning has a tendency to be more subjective but may also be seen as a variation of the first. For example, anything can be expressed using propositional logic, the problem being that an exponential number of statements may be needed to say it. First order predicate calculus can immediately get rid of the exponential number of statements (perhaps at the loss of clarity). It is the second meaning of expressivity, that is, how naturally can something be stated, that is important in this paper.

A problem with greater expressivity is the increased potential for intractability. This was seen by the problem of trying to define languages that would apply to many domains. The language defined here attempts to move from the more expressive world of first order logic and a basic knowledge base management system model toward an OPS5 world of more efficiency without losing any expressivity (perhaps even gaining some). Figure 2 depicts this goal graphically.

2.1 The Knowledge Principle

To beat the problem of intractability, domain heuristics are used. Raj Reddy's fifth principle states: *Knowledge eliminates the need for search* [16] (see also [10]). The success of the domain dependent expert systems was almost exclusively a result of this. This implies that a language that can adequately represent heuristic knowledge (both declarative and procedural) provides the power to eliminate much of the search, making problems tractable once again.

On the other hand, Raj's fourth principle states: *Search compensates for the lack of knowledge*. This implies that in the cases where knowledge is not present, general search strategies may be used to fall back on. Thus, the specification of domain knowledge and control knowledge of the domain tremendously enhances the tractability of the knowledge representation and interpretation problem (see [3] as a classic example of using heuristics to beat the tractability problem).

2.2 Options for Managing Tractability

As a language becomes more expressive, the problem of tractability becomes more important. There are two aspects to managing this problem. The first aspect is to manage tractability by providing heuristic adequacy in the language [21]. This method manages tractability by using heuristics in place of search; providing direct paths to a solution. The second method is by providing efficiency in the language itself, for example, the RETE network in OPS5.

Tractability must also be managed when search is used to compensate for the lack of heuristics. There are at least three options available for managing the expressivity versus tractability problem when using general search methods [11]. First Order Logic (FOL) will be used to represent complete expressivity. Most will agree that practically anything can be represented in FOL. Furthermore, resolution theorem provers have been built which are sound and complete and will derive a proof in FOL if the proof exists in the theory. The only problem is that the theorem prover may take an exponentially long period of

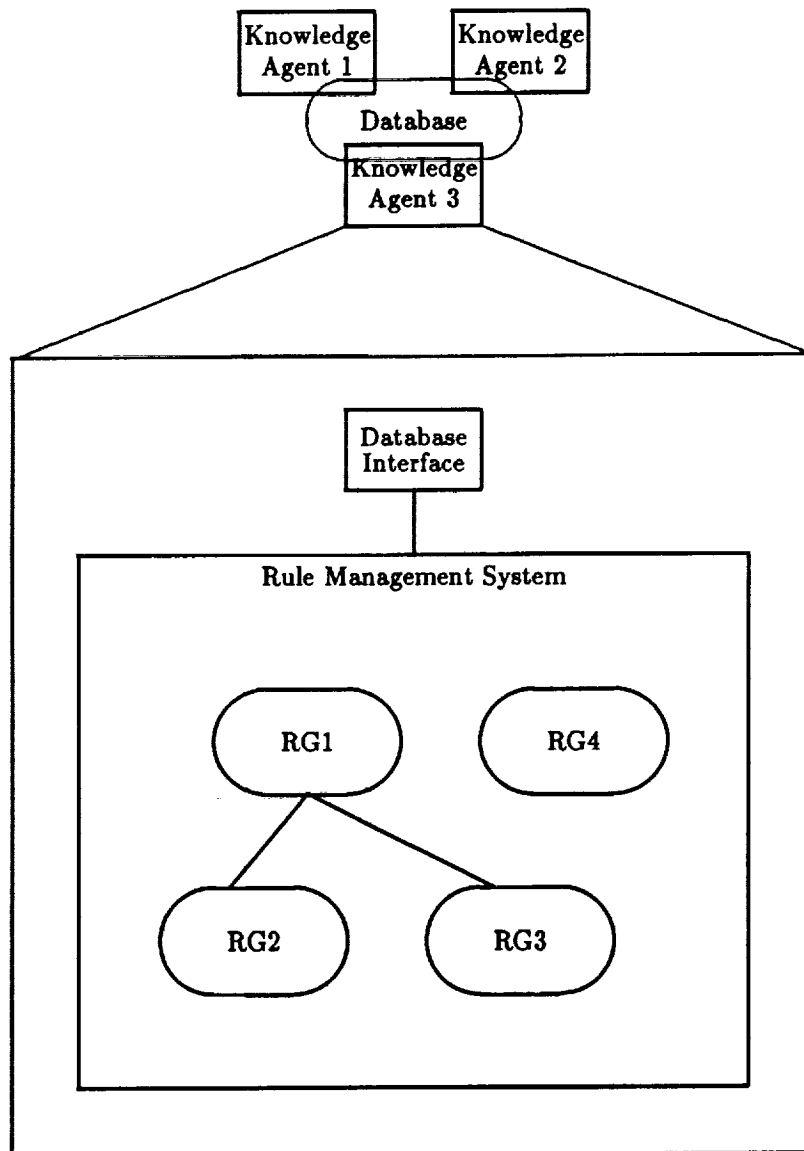


Figure 1: Rule Management System Architecture

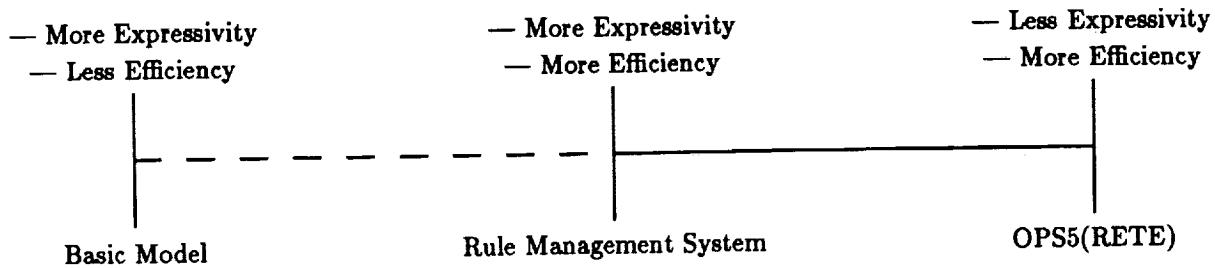


Figure 2: Expressivity versus Efficiency

time deriving the proof. It may not even return at all if the MINUS lisp :: first (rule-result)] ; proof does not exist. The alternatives are to: One, restrict what the language can express (e.g. allow only conjunction) or two, restrict the definition of implication (or derivability). One way of doing this is to define an implication operator that is sound but incomplete. Finally, a third alternative is to use defaults and assumptions. Defaults and assumptions are probably the least understood option. In fact, it is even possible that the use of defaults and assumptions (introducing non-monotonicity) may be more intractable than monotonic reasoning.

Common mechanisms for restricting the expressivity of the representation language are by the use of databases, logic programs, versions of FOL, expert system languages, etc. The power of machine learning programs are also usually a function of what they can represent. The representation language in combination with the domain knowledge defines the size of the search space [8]. Limits on knowledge expression determine bounds on the size of the search space. Thus, this restriction on knowledge representation is a mechanism for reducing the size of the search space.

The most common way of restricting implication is to define methods of inference that are sound but not complete. One of the easiest ways to do this is to not allow backtracking. For example, in the area of expert systems, there may be more than one rule in the conflict set. During forward chaining only one rule may be allowed to fire from the conflict set. Once this rule is fired no backtracking is allowed to try the other rules. Therefore, this strategy can be seen to be sound but not complete. All possible solutions may not be found.

Finally, defaults and assumptions may be used to infer knowledge without having to explicitly derive it using an inference strategy. Knowledge is available due to inheritance and other mechanisms. This approach helps tractability by eliminating the need to infer defaulted and assumed knowledge. On the other hand, managing defaults and assumptions (as well as inheritance with exceptions) may cause some intractability problems of its own.

3 Language Design and Implementation for Domain Independence

The following sections describe in some detail the design and implementation of the rule management system language and support.

3.1 The Database

The first level of support is the database. The database is viewed as a plug compatible module. This provides the ability to take advantage of existing databases and database management mechanisms such as ORACLE or INGRES, for example.

The rule management system takes a tuple space view of the world and requires that the database represent them. A tuple is an ordered sequence of (possibly) typed fields. Conceptually, a tuple may be

thought of as an object that exists independent of the process that created it. This implies that a place to store it is needed if it is to live a life independent of processes used to create and destroy it. The tuple space is sufficient for representing FOL and also sufficient for representing at least relational databases [17].

The tuple space has also been proposed as a mechanism for supporting distributed processing [4]. There are three basic mechanisms that a process may perform on tuples in the tuple space: **IN**, **OUT**, and **READ**. The tuple fields in the **IN** and **READ** operations may optionally contain variable arguments for matching. **IN** and **READ** block until the tuple is present in the database. **IN** subsequently removes the tuple from the database while **READ** does not. **OUT** installs a tuple in the database. The benefit to distributed processing that the tuple space provides is the ability to add processes to the environment and have them communicate with existing processes without first having to encode knowledge about existing processes directly into their formalism. It has also been proposed that these basic tuple space operations are sufficient for supporting other communication mechanisms such as those defined in contract nets and Actors. On the other hand, the use of **IN** must be judicious in order to support the addition of other processes that may also need to use the tuple. See [4] and [13] for more details.

The database provides three operations similar to the tuple space operations; these are: **STORE**, **RETRIEVE/MATCH**, and **DB-REMOVE**. The major difference between these operators and the tuple space operators **IN**, **READ**, and **OUT**, is that these do not block. Another difference is that while **IN** and **READ** will non-deterministically select one of multiple matching tuples, **MATCH** will return all of them (**RETRIEVE** will behave non-deterministically). Although the database operations defined here are not identical to the tuple space operations in the LINDA model, it is probably sufficient for handling distributed processing protocols as the LINDA model [4].

The implementation of the database also supports a form of integrity constraints and a restricted version of views. An integrity constraint is defined by a tuple. Fields of the tuple may contain one of four items: An actual field descriptor (formal argument), a variable (to match an argument), a type specifier—meaning that the field may match tuples with the corresponding field of the same type, and finally, a type declarator—meaning that for all tuples matched on the other three options, this field must be of the declared type. Views provide different databases for storage and retrieval. This is quite different to the traditional use of views in most databases. What this provides is a way to organize data into more logical groupings. Currently, the use of integrity constraints is global across all views, in the future this should use the view mechanism just as any other storage or retrieval operation does.

3.2 The Database Interface

The database interface is the module that links the database to the rule management system and vice versa. The interface provides the necessary hooks for proper accessing and notification of data in the database and of interest to the rule management system. The database interface provides the operations **STORE!**, **REMOVE!**, **MATCH!** and **RETRIEVE!**. It also provides a data and procedure abstraction mechanism—a frame system [9, 14, 20]. Frames are used to extend the knowledge representation language of the various tools for supporting complex domains requiring abstraction mechanisms.

Frames are an abstract organization of data into conceptual units. In this definition data may also be procedure if it is subsequently executed. A frame may have any number of slots. Frames may be defined as children of multiple parents (making inheritance potentially more complex) and may also have code attached to them that is executed whenever a new instance of the frame or one of its children is created. Slots have six optional aspects. The most used aspect is the **:value** aspect. This aspect is where a value for the slot is located. The **:if-needed** aspect is used to store code that is executed if a slot value is asked for. The **:if-added** aspect is used to store code that is executed whenever the slot gets a new value. There are two aspects that are used to constrain the value of the slot. The **:constraint** aspect is used to store code that checks if an added value passes the constraint. Since this is user-defined code that is used here there is no restriction on what it may do, only that it return a true or false status indicating the result of the constraint. The **:mustbe** aspect constrains the value to be one of a list of

formal values or frames. Finally, the `:distribution` aspect is used to determine if the value of the slot is for global distribution, accessible to all knowledge agents, or only for local usage.

Frame data is stored in the database as 5-tuples of the form: `(frame <name> <slot> <aspect> <value>)`. Obviously, there is no restriction on the value aspects, they can be any normal data type as well as executable code. Facts are stored in the database as 4-tuples: `(fact <name> :value <value>)`.

Frame inheritance information must also be stored in the database so that inheritance over frames may take place in response to different knowledge agents requests for data values. The current implementation does not yet store this data in the database.

It is the responsibility of the database interface to both store and retrieve information and to notify the rule management system of changes to data made by other knowledge agents. Thus if knowledge agent 1 makes a change to the value of a fact and knowledge agent 2 uses that fact on the left hand side (LHS) of a rule, it is the responsibility of the database interface (and distributed database) to notify knowledge agent 2 of the changed fact.

3.2.1 User Objects Example

As an example of using the frame system to represent user objects, the rule group and knowledge base objects are shown here as they are defined for a large fault diagnosis knowledge base.

```
(frame :name knowledge-base
  :slots ((rule-groups :value nil)
    (agents :value nil)
    (name :value nil)))

(frame :name rule-group
  :slots ((rules :value nil)
    (name)
    (window)
    (quantified-vars)
    (rg-var)
    (plan)
    (plan-state)
    (plan-table :value nil)
    (viable-set :value nil)
    (not-yet-viable-set :value nil)
    (fire-set :value nil)
    (local-variables)
    (rules-fired :value nil)
    (conflict-set :value nil)
    (tickle-set :value nil)
    (satisfied-set :value nil)
    (unsatisfied-set :value nil)
    (cant-fire-set :value nil)
    (fired-set :value nil)
    (untickled-set :value nil)
    (*lhs-tickled-queue* :value nil)
    (*rhs-tickled-queue* :value nil)
    (termination-condition :value nil)
    (rules-with-dynamic-lhs-patterns :value nil)
    (rules-with-dynamic-rhs-patterns :value nil)
    ;; each el: (plan-state rules)
    (backtrack-stack :value nil)
    (control-strategy :value #'default-control-strategy)
    (conflict-resolution-strategy :value
      #'default-conflict-resolution-strategy)
    (execute :value #'execute1)
  ))
```

3.3 The Rule Management System

The rule management system consists of two layers. The base layer is the rule language. This language defines what rules are and how they are evaluated and interpreted. The second layer defines a knowledge base management system (KBMS) that is built on top of the rule language. While the KBMS is a layer on top of the rule language, it does not add anything to it; it is completely defined in terms of the rule language³. The correct way to think of this is that the rule language is the basic level of expression for the knowledge engineer. The knowledge engineer then writes a number of rules that define the KBMS. This KBMS is like another shell that may be instantiated with a domain application, thereby defining an expert system.

The rule language and the KBMS work hand in hand. The KBMS is a predefined object available to the knowledge engineer. However, the KBMS definition may be easily modified and extended for special applications. Furthermore, control strategies and conflict resolution strategies may be defined using the rule language itself or by writing procedures (a kind of compiled form of the strategy). These user definable strategies in combination with the expressibility of the rule language provide the power to the rule management system for application to a wide variety of domains.

The KBMS defines a knowledge base to consist of a number of rule groups as well as domain knowledge. Each rule group provides mechanisms for defining aspects of the inference strategy using either further rule groups or user-defined functions. A rule group also has a mechanism for specifying the level of determinism desired over rule execution [1], [7]. A mechanism for explicitly stating a level of deterministic control over a set of rules is one of the things missing from many expert system shells. Without this, the only way to get the necessary control is by inserting control knowledge into the rules themselves—where it does not belong. By explicitly stating the difference between control over knowledge and the knowledge itself, the maintenance problem becomes easier. The way control is added is by examining the inference cycle of an expert system. Basically, the rule group inference strategy consists of a match, evaluate, and fire cycle. The match phase is supported by the database interface which automatically queues up potential rules on the tickled-queues of the rule group. The evaluate phase then checks the rules on the tickled-queues to determine which ones belong in the conflict set. Finally, one rule is selected and fired. This approach is completely non-deterministic in selecting which rules get fired from the conflict set. In between the match and evaluate phases a control phase is added. This control phase consists of the definition of a regular expression that defines which rules may be examined and possibly fired next. In this implementation the transition table derived from the regular expression is used instead of the regular expression itself. This is much simpler from the user's perspective. For the user defining a number of rules and wanting to insert some control over them, it is easier to specify a transition table over the rules than it is to define the regular expression that the transition table can be derived from. Furthermore, it is easier to maintain a transition table during rule group modification.

The language of the rule management system is very complex. It supports more traditional data access such as facts and frames. It supports procedural execution directly from rules in the form of messages. This is how the KBMS is initially linked to the rule language. A default execution strategy, including control strategy and conflict resolution strategy exists which may be started from a rule using the message form. The rule language also supports a form of typed quantification.

3.3.1 Heuristic Control Example

An example of the power of the rule language is the application of a control strategy for a KBMS. The following example defines a rule group solely for performing the control strategy aspect of a forward chainer for a KBMS. The point of this example is the ease with which these heuristics may be defined and applied.

³ However, to make this work efficiently there has been some optimization. The rule language makes certain assumptions about the existence of certain KBMS domain constructs. An example of this is the assumed existence of a rule-group frame with a "rules" slot and a "lhs-tickled-queue" slot.


```

Rule-Group : freight-cs
;;
;; The first rule to be run is CS-Rule1.
;; Following this we only allow CS-Rule2.
;; After CS-Rule2, we non-deterministically only allow CS-Rule3-5.
;;
CONTROL : ((start (CS-Rule1))
           (CS-Rule1 (CS-Rule2))
           (CS-Rule2 (CS-Rule3 CS-Rule4 CS-Rule5))
           (CS-Rule3 (CS-Rule3 CS-Rule4 CS-Rule5))
           (CS-Rule4 (CS-Rule3 CS-Rule4 CS-Rule5)))
;;
;; The variable that we quantify over is rule-group.
;; This variable will be bound to
;; the rule group that this control strategy is applied to.
;;
RG-VAR : rule-group
;;
;; CS-Rule1 is an initialization rule.
;; It simply makes sure that we start with an
;; empty list.
;;
CS-Rule1
::>
[ rule-result-list = empty ]
;
;;
;; CS-Rule2 evaluates all the rules in the viable-set of the rule-group and
;; records the result in the rule-result-list. Each result takes the form of
;; (rule result), the result can be one of: :ok, :ng, or :missing-patterns
;;
CS-Rule2
FOR ALL rule in viable-set of rule-group
< ::>
[ rule-result-list =
  rule-result-list PLUS lisp :: my-evaluate ( rule ) ] >
;
;;
;; CS-Rule3 adds all the rules which evaluated to :ok to the conflict set of
;; the rule group. It also removes them from the viable set.
;;
CS-Rule3
FOR ALL rule-result in rule-result-list
WHERE [ lisp :: second ( rule-result ) = :ok ]
< ::>
[ conflict-set of rule-group = conflict-set of rule-group
  PLUS lisp :: first ( rule-result ) ]
[ viable-set of rule-group = viable-set of rule-group
  MINUS lisp :: first ( rule-result ) ] >
;
;;
;; Similarly, CS-Rule4 watches for rules with a :ng result. These that have an
;; else are fired (via my-interpret-else) and also removed from the viable set.
;;
CS-Rule4
FOR ALL rule-result in rule-result-list
WHERE [ lisp :: second ( rule-result ) = :ng ]
< ::>
[ lisp :: my-interpret-else ( lisp :: first ( rule-result ) ) ]
[ viable-set of rule-group = viable-set of rule-group
  MINUS lisp :: first ( rule-result ) ] >
;
;;
;; CS-Rule5 watches for the rules that had :missing-patterns and simply removes

```

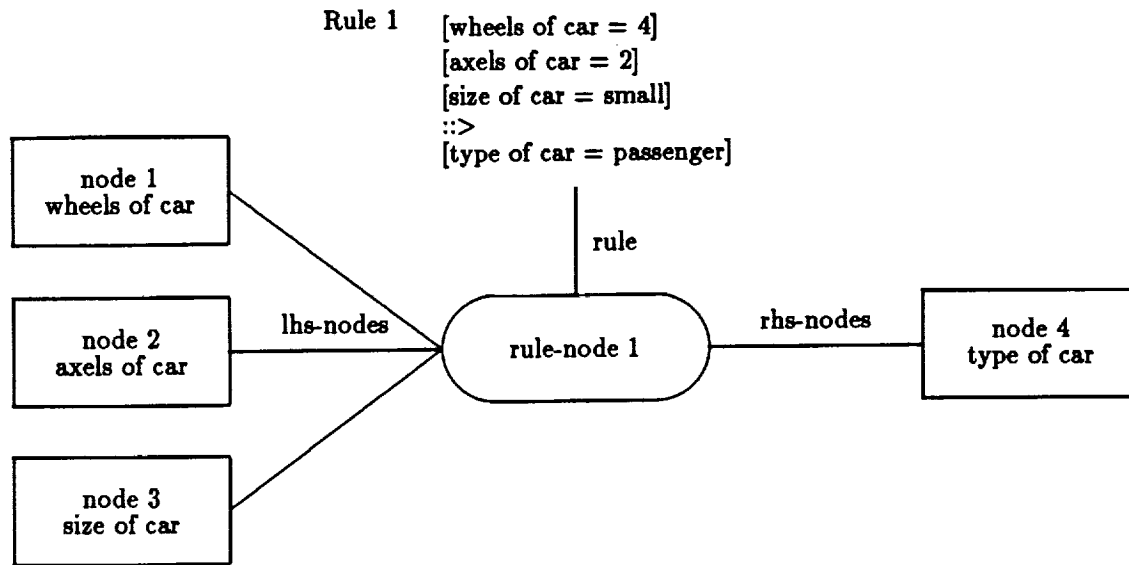


Figure 3: Rule Constraint Network Example

```

%% them from the viable set of the rule group.
%%
CS-Rule5
FOR ALL rule-result in rule-result-list
  WHERE [ lisp :: second ( rule-result ) = :missing-patterns ]
  < ::>
    [ viable-set of rule-group = viable-set of rule-group
      MINUS lisp :: first ( rule-result ) ] >
:
%%
%% Finally, this rule group is done when the viable set is empty.
%%
[ viable-set of rule-group = empty ]

```

3.4 Tractability and the Rule Management System

Considering the generality and expressivity of the rule language and the KBMS, efficiency has the potential of being forgotten, both in development and implementation. There are three ways of maintaining tractability in the rule language.

The first way of maintaining tractability is to manage the match phase of the inference cycle efficiently. The conceptual definition of the match phase is to check each rule and evaluate its LHS and if it is :ok then to queue it up for later interpretation. Obviously, this is also the slowest approach. The step this implementation takes is to compile rules into a rule constraint network that maintains rules in a form more suited for recognizing when data becomes available that has the potential of satisfying the LHS of a rule. To be specific, all the reference variables of the LHS of a rule must have values in order to determine if the LHS is satisfied. The rule constraint network represents variables as nodes in the network and rules as rule-nodes (see Figure 3). As variables get values, the nodes representing the variables are triggered. All the rule-nodes connected to the node are then triggered. These rule-nodes are then checked to see if all the nodes representing the LHS variables have values. If so, the rule represented by the rule-node is then put on the appropriate queue. As can be seen, this method allows rules to be queued that may not be satisfied. The rule constraint network only checks to see if variables have values, not if they have the correct value. Thus the rules on the tickled queues must still be evaluated for satisfaction.

Alternatively, rules could be compiled more completely so that variables are checked to see if they

have the right value. The complexity of the rule language implemented here makes this a difficult task and it has not been determined that it would be cost effective. Languages with less expressivity can be completely compiled much more easily (such as OPS5 using the RETE network [5] [6]).

The second way of maintaining tractability is by providing code for inference strategies instead of providing declarative rules defining the inference strategy. This is making use of the compiled versus interpreted option. This means that a programmer fluent in the rule language and its implementation needs to be available for the knowledge engineering task. However, considering the expressivity of the rule language, this may be a cost effective option. It is very easy for the programmer to express what he wants without having to go into contortions over representational limits.

The third way to maintain tractability is by the effective use of knowledge. This is to make use of Raj Reddy's fifth principle: *Knowledge eliminates the need for search*. In other words, the domain to be represented needs to be analyzed for maximum efficiency in terms of knowledge organization. For example, I can have two rule groups; one rule group forward chains on various data and computes a value for the variable *diagnosis*. The other rule group then uses the value of *diagnosis* to output the results to the user. If there are 50 rules in each rule group that use the variable, then 100 rules are triggered whenever the value of the variable changes. If both of these rule groups are made to be sub-rule groups of a control rule group, the first rule group can compute a value for *diagnosis*. The control rule group can then set the value for the variable *the-diagnosis* as the value of *diagnosis*. *the-diagnosis* is then used by the second rule group instead of *diagnosis*. Now only 50 rules get triggered at any time.

The representation of knowledge should make maximum use of divide and conquer principles of knowledge organization. Another approach along the same lines is to provide strong heuristic knowledge to eliminate the need for search. This can come in many forms including domain guided inference strategies over rule groups as well as judicious use of control over the execution of rules in a rule group (i.e. the transition table method of control over rules).

4 conclusion

The rule management system language presented here has been shown to be both general and potentially efficient. It can be applied to many domains by proper specification — making use of the object oriented nature of the language. This generality allows it to be applied to many domains including learning, planning, and model-based reasoning as well as expert systems.

5 Acknowledgements

Thanks to Barry Ashworth and Bryan Walls for useful comments and criticisms.

This work was performed by Martin Marietta Astronautics Group under contract number NAS8-36433 to NASA, George C. Marshall Space Flight Center, Huntsville, Alabama.

References

- [1] A. B. Baskin. Combining deterministic and non-deterministic rule scheduling in an expert system. In *AAMSI*, 1986.
- [2] Bruce G. Buchanan and Richard O. Duda. *Principles of Rule-Based Expert Systems*. Technical Report HPP-82-14, Stanford Heuristic Programming Project, 1982.
- [3] Bruce G. Buchanan and Edward A. Feigenbaum. Dendral and meta-dendral: their applications dimension. In *Readings in Artificial Intelligence*, pages 313-322, Morgan-Kaufmann Publishing Co., 1981.

- [4] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4), 1989.
- [5] Charles L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1), 1982.
- [6] Charles L. Forgy and Susan J. Shepard. Rete: a fast match algorithm. *AI Expert*, January 1987.
- [7] M.P. Georgeff. Procedural control in production systems. *Artificial Intelligence*, 18:175-201, 1982.
- [8] D. Haussler. Bias, version spaces, and valiant's learning framework. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 324-336, 1987.
- [9] P.J. Hayes. The logic of frames. In B.L. Webber and Nils J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 451-458, Morgan Kaufmann, 1981.
- [10] Douglas B. Lenat and Edward A. Feigenbaum. On the thresholds of knowledge. In *International Workshop on Artificial Intelligence for Industrial Applications*, 1988.
- [11] Hector J. Levesque. Knowledge representation and reasoning. *Annual Reviews of Computer Science*, 1986.
- [12] C.L. Liu. *Elements of Discrete Mathematics*. McGraw Hill Book Co., 1985.
- [13] Satoshi Matsuoka and Satoru Kawai. Using tuple space communication in distributed object-oriented languages. In *OOPSLA*, 1988.
- [14] M. Minsky. A framework for representing knowledge. In P. Winston, editor, *The Psychology of Computer Vision*, pages 211-277, McGraw-Hill, 1975.
- [15] A. Newell, J.C. Shaw, and H.A. Simon. A general problem-solving program for a computer. In *Information Processing: Proceedings of the International Conference on Information Processing*, pages 256-264, UNESCO, Paris, 1960.
- [16] Raj Reddy. Presidential Address at the American Association for Artificial Intelligence Conference, 1988.
- [17] R. Reiter. Towards a logical reconstruction of relational database theory. In M. Brodie, J. Mylopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling*, Springer-Verlag, 1984.
- [18] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23-41, January 1965.
- [19] Mark Stefik, Jan Aikins, Robert Balzer, John Benoit, Lawrence Birnbaum, Frederick Hayes-Roth, and Earl Sacerdoti. The organization of expert systems. *Artificial Intelligence*, 18, 1982.
- [20] Mark Stefik and Daniel G. Bobrow. Object-oriented programming: themes and variations. *The AI Magazine*, 40-62, 1986.
- [21] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, 1988.

**KNOWLEDGE MANAGEMENT: AN ABSTRACTION OF
KNOWLEDGE BASE AND DATABASE MANAGEMENT SYSTEMS**

**A PAPER PRESENTED AT THE AI SYSTEMS IN GOVERNMENT CONFERENCE
IN MAY 1990.**

Knowledge Management: An Abstraction of Knowledge Base and Database Management Systems

Joel D. Riedesel
Martin Marietta Astronautics Group
P.O. Box 179, MS: S-0550
Denver, Co. 80201
jriedesel@den.mmc.com

May 22, 1990¹

Abstract

Artificial Intelligence application requirements demand powerful representation capabilities as well as efficiency for real-time domains. Many tools exist, the most prevalent being expert systems tools such as ART, KEE, OPS5, and CLIPS. Other tools just emerging from the research environment are truth maintenance systems for representing non-monotonic knowledge, constraint systems, object oriented programming, and qualitative reasoning. Unfortunately, as many knowledge engineers have experienced, simply applying a tool to an application requires a large amount of effort to *bend* the application to fit. Much work goes into supporting work to make the tool integrate effectively.

KNOMAD, a Knowledge Management Design System, described here, is a collection of tools built in layers. The layered architecture provides two major benefits; the ability to flexibly apply only those tools that are necessary for an application, and the ability to keep overhead, and thus inefficiency, to a minimum. KNOMAD is designed to manage many knowledge bases in a distributed environment providing maximum flexibility and expressivity to the knowledge engineer while also providing support for efficiency.

¹This work was performed under contract NAS8-36433 to NASA, George C. Marshall Space Flight Center

1 Introduction

Early Artificial Intelligence (AI) work centered around domain *independent* representation and reasoning tools. Examples include General Problem Solver [23] and general purpose theorem provers such as resolution theorem proving ([27]). These early approaches included powerful knowledge representation languages and sound and complete search methods. However, as large problems were applied these early programs proved to be intractable.

To manage the complexity and tractability of large domains there was a shift to domain *dependent* systems, primarily expert systems. These domain dependent systems could then apply domain heuristics instead of general search and restricted knowledge representation languages with inherent bias to reduce the search space ([4, 29]). Although these domain dependent expert systems worked quite well, expert system technology and knowledge representation of various expert system shells did not transfer well from one domain to the next. Witness the diversity of expert system shells currently existing in the commercial marketplace. The number of shells residing in the university research labs are probably an order of magnitude larger. This diversity is a result of two main restrictions put on the knowledge representation language: one, the restriction on knowledge that can be described using the language of the shell and two, the restriction on mechanisms of inference over the knowledge. Because of these restrictions, it became very important to analyze a domain very carefully to be certain that it could be represented in an existing shell. If a good match was not found two approaches could be taken: the shell could be used anyway and the application *bent* to fit it, or a home-grown shell that better matched the application could be built. These domain dependent solutions turned out to be very tractable while trading generality. This does not only apply to expert system shells, but to software and AI tools in general ([18]).

Recently there has been a better understanding of the issues of search and heuristics and how they can be used together to solve complex problems. An example system is SOAR which does large amounts of search initially to solve a problem but learns from the solutions found. The learned solutions may then be applied as heuristics for future problems ([16]). The point of all this being that heuristics can eliminate the need for search while search compensates for the lack of heuristics ([24]). To make use of this concept in AI tools, the tools must be capable of supporting the representation of domain dependent heuristics in a general fashion. The tools need to have heuristic adequacy (see [34]) as well as general search mechanisms. They need to be very general with the intent that the knowledge engineer specializes or instantiates the tool to the specific application, thus making the tool efficient. The current generation of AI tools do not support this sort of engineering environment².

Real world problems need more than just heuristic adequacy in a tool. They need a collection of tools for representing and reasoning about the various aspects that make up the problem. No one tool will likely be able to manage a large scale, real world problem ([18, 17]). Furthermore, a collection of tools ought to be expandable and modular. They ought to be able to reason about the same data and share data easily.

The Knowledge Management Design System (KNOMAD), described here, more properly belongs to the next generation of AI tools in the applications world. It may be considered as a knowledge engineering *environment*. It provides a set of tools including rule management, database, constraint system, frame system, model support, and so forth. These tools are organized in a modular fashion so that more tools may be added and other tools may be removed or substituted as the application requires. The rule management system tool, for example, provides a powerful rule language while also providing mechanisms to keep the tool efficient.

² Although CL ([33]) is a step in the right direction.

2 Artificial Intelligence Application Requirements

The wide variety of available applications impose a large and diverse set of requirements on necessary tools for encoding knowledge about them as well as performing inference over them. Furthermore, most applications require more than one tool to represent the necessary knowledge. The following is a list of tools, some set of which will be required by any application:³

- A model building tool
- A database for both distributed and local data
- Procedural representation and execution
- Abstraction for both data and procedures
- Scripts and frames
- Object oriented programming
- Constraint posting and propagation
- Analytic, qualitative, and quantitative reasoning
- Semantic nets
- Rule management

In addition, representation of both temporal and non-monotonic knowledge may need to be mixed in with some of the above tools as an application requires.

Combinations of these tools may be put together to build applications in planning and prediction domains, diagnosis domains, and control domains, for example.

2.1 Expressibility and Efficiency

Expressivity has at least two meanings that may be distinguished. One meaning of expressiveness asks whether a language allows something to be said. In this meaning various languages can be compared in terms of formal language theory and classified as Type 0, 1, 2, or 3 ([20]). We find that most computer languages are Turing equivalent and therefore it becomes moot to compare languages on the simple basis of whether or not something can be said in them. The other meaning of expressiveness asks how easy can it be said, how clear or understandable is it using the language? This meaning has a tendency to be more subjective but may also be seen as a variation of the first. For example, anything can be expressed using propositional logic, the problem being that an exponential number of statements may be needed to say it. First order predicate calculus can immediately get rid of the exponential number of statements (perhaps at the loss of clarity). It is the second meaning of expressivity, that is, how naturally can something be stated, that is important here.

As a language becomes more expressive, the problem of tractability becomes more important. There are two aspects to managing this problem. The first aspect is to manage tractability by providing heuristic adequacy in the language ([34]). This method manages tractability by using heuristics in place of search; providing direct paths to a solution. The second method is by providing efficiency in the language itself, for example, the RETE network in OPS5.

There are at least three options available for managing the expressivity versus tractability problem when using general search methods ([19]). First Order Logic (FOL) will be used to represent complete expressivity. Most will agree that practically anything can be represented in FOL. Furthermore, resolution theorem provers have been built which are sound and complete and will derive a proof in FOL if the

³See Hayes-Roth ([14]) for a more complete discussion of objects that are required by AI tools.

proof exists in the theory. The only problem is that the theorem prover may take an exponentially long period of time deriving the proof. It may not even return at all if the proof does not exist. The alternatives are to: One, restrict what the language can express (e.g. allow only conjunction) or two, restrict the definition of implication (or derivability). One way of doing this is to define an implication operator that is sound but incomplete. Finally, a third alternative is to use defaults and assumptions. Defaults and assumptions are probably the least understood option. In fact, it is even possible that the use of defaults and assumptions (introducing non-monotonicity) may be more intractable than monotonic reasoning.

Common mechanisms for restricting the expressivity of the representation language are by the use of databases, logic programs, versions of FOL, expert system languages, etc. The power of machine learning programs are also usually a function of what they can represent. The representation language in combination with the domain knowledge defines the size of the search space ([12]). Limits on knowledge expression determine bounds on this space. Thus, this restriction on knowledge representation is a mechanism for reducing the size of the search space.

The most common way of restricting implication is to define methods of inference that are sound but not complete. One of the easiest ways to do this is to not allow backtracking. For example, in the area of expert systems, there may be more than one way to derive a goal during backward chaining. By using a beam search some of the possibilities may be thrown out and never considered by backtracking. Therefore, this strategy can be seen to be sound but not complete. All possible solutions may not be found (or even no solution).

Finally, defaults and assumptions may be used to infer knowledge without having to explicitly derive it using an inference strategy. Knowledge is available due to inheritance and other mechanisms. This approach helps tractability by eliminating the need to infer defaulted and assumed knowledge. On the other hand, managing defaults and assumptions (as well as inheritance with exceptions) may cause some intractability problems of its own.

3 The KNOMAD Architecture

To solve the problems of flexibility and expressivity the KNOMAD architecture was developed. While trying to be both an expressive and powerful language, efficiency was also a primary issue. The KNOMAD architecture is a layered architecture as shown in Figure 1. The central component is the database, a place for storing working memory data, for transferring and sharing data between knowledge agents, and for storing long term data. The database is designed as a module and may be implemented as a distributed database. A distributed database may then be used to support multiple cooperating knowledge agents, each residing in different physical locations. The next layer is an interface to the database that provides a frame system for abstracting both data and procedure as well as a mechanism for storing simple facts. The top layer is where various tools are defined and implemented. All the tools make use of the same data representation and may easily share data across domains and functions.

The architecture will be discussed in some detail in the following sections including aspects of implementation and efficiency.

3.1 The Database

The first level of support for KNOMAD is the database. The KNOMAD architecture views the database as a plug compatible module. This provides the ability to take advantage of existing databases and database management mechanisms such as ORACLE or INGRES, for example.

KNOMAD takes a tuple space view of the world and requires that the database represent them. A tuple is an ordered sequence of (possibly) typed fields. Conceptually, a tuple may be thought of as an object that exists independent of the process that created it. This implies that a place to store it is needed if it is to live a life independent of processes used to create and destroy it. The tuple space is sufficient for representing FOL and also sufficient for representing at least relational databases ([25]).

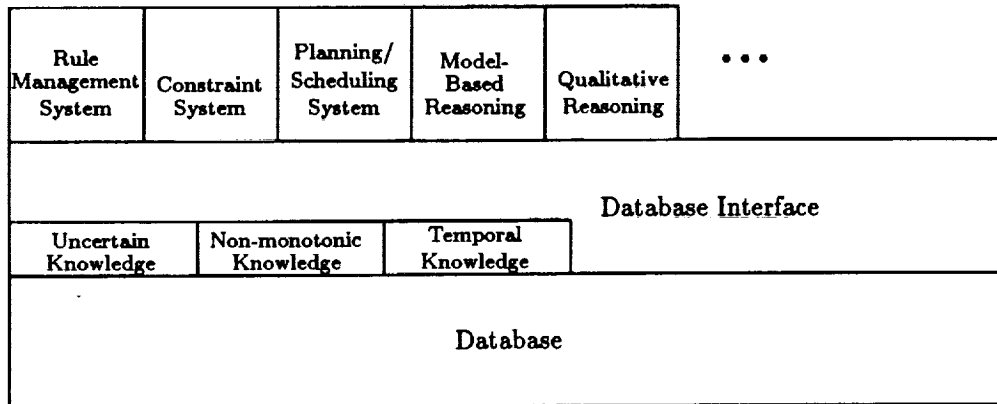


Figure 1: KNOMAD Layered Architecture

The tuple space has also been proposed as a mechanism for supporting distributed processing (the LINDA model [5]). There are three basic mechanisms that a process may perform on tuples in the tuple space (as in the LINDA model): **IN**, **OUT**, and **READ**. The tuple fields in the **IN** and **READ** operations may optionally contain variable arguments for matching. **IN** and **READ** block until the tuple is present in the database. **IN** subsequently removes the tuple from the database while **READ** does not. **OUT** installs a tuple in the database. The benefit to distributed processing that the tuple space provides is the ability to add processes to the environment and have them communicate with existing processes without first having to encode knowledge about existing processes directly into their formalism. It has also been proposed that these basic tuple space operations are sufficient for supporting other communication mechanisms such as those defined in contract nets and Actors. On the other hand, the use of **IN** must be judicious in order to support the addition of other processes that may also need to use the tuple. See [5] and [21] for more details.

The database provides three operations similar to the tuple space operations; these are: **STORE**, **RETRIEVE/MATCH**, and **DB-REMOVE**. The major difference between these operators and the tuple space operators **IN**, **READ**, and **OUT**, is that these do not block. Another difference is that while **IN** and **READ** will non-deterministically select one of multiple matching tuples, **MATCH** will return all of them (**RETRIEVE** will behave non-deterministically). Although the database operations defined here are not identical to the tuple space operations in the LINDA model, they are probably sufficient for handling distributed processing protocols as in the LINDA model.

This view of the database allows for many different implementations of the database to be used, including distributed databases. A distributed database would provide the ability to support distributed knowledge agents defined and managed using KNOMAD. Their communication is then supported by the tuple space mechanism as implemented in the distributed database.

The implementation of the database for KNOMAD also supports a form of integrity constraints and a restricted version of views. An integrity constraint is defined by a tuple. Fields of the tuple may contain one of four items: An actual field descriptor (formal argument); a variable (to match an argument); a type specifier, meaning that the field may match tuples with the corresponding field of the same type; and finally, a type declarator, meaning that for all tuples matched on the other three options, this field must be of the declared type. Views provide different databases for storage and retrieval. This is quite different to the traditional use of views in most databases. What this provides is a way to organize data into more logical groupings. Currently, the use of integrity constraints is global across all views, in the future this should use the view mechanism just as any other storage or retrieval operation does.

3.2 The Database Interface

The database interface is the module that links the database to KNOMAD and KNOMAD to the database. The interface provides the necessary hooks for proper accessing and notification of data in the database and of interest to KNOMAD. The database interface provides the operations **STORE!**, **REMOVE!**, **MATCH!** and **RETRIEVE!**. It also provides a data and procedure abstraction mechanism, a frame system ([13, 22, 30]). Frames are used to extend the knowledge representation language of the various tools for supporting complex domains requiring abstraction mechanisms.

Frames are an abstract organization of data into conceptual units. In this definition data may also be procedure if it is subsequently executed. A frame may have any number of slots. Frames may be defined as children of multiple parents (making inheritance potentially more complex) and may also have code attached to them that is executed whenever a new instance of the frame or one of its children is created. Slots have six optional aspects. The most used aspect is the **:value** aspect. This aspect is where a value for the slot is located. The **:if-needed** aspect is used to store code that is executed if a slot value is asked for. The **:if-added** aspect is used to store code that is executed whenever the slot gets a new value. There are two aspects that are used to constrain the value of the slot. The **:constraint** aspect is used to store code that checks if an added value passes the constraint. Since this is user-defined code that is used here there is no restriction on what it may do, only that it return a true or false status indicating the result of the constraint. The **:mustbe** aspect constrains the value to be one of a list of formal values or frames. Finally, the **:distribution** aspect is used to determine if the value of the slot is for global distribution, accessible to all knowledge agents, or only for local usage.

Frame data is stored in the database as 5-tuples of the form: (frame <name> <slot> <aspect> <value>). Obviously, there is no restriction on the value aspects, they can be any normal data type as well as executable code. Facts are stored in the database as 4-tuples: (fact <name> :value <value>).

Frame inheritance information must also be stored in the database so that inheritance over frames may take place in response to different knowledge agents requests for data values. The current implementation does not yet store this data in the database.

It is the responsibility of the database interface to both store and retrieve information and to notify the various tools of changes to data made by other knowledge agents. Thus if knowledge agent 1 makes a change to the value of a fact and knowledge agent 2 uses that fact on the left hand side (LHS) of a rule, it is the responsibility of the database interface (and distributed database) to notify knowledge agent 2 of the changed fact and possibly activate the rule for forward chaining.

3.3 The Tool Layer

The tool layer is where the various reasoning tools are defined. This is where the rule management system, the constraint system, the planning system, the model-based reasoning system, the qualitative reasoning system, etc. are defined and implemented. Each of these various tools may define different language representations and certainly define different reasoning mechanisms. However, all the tools must use the underlying database and interface for storage and retrieval of data ([15, 28, 2, 6, 32, 1, 31, 8, 7]).

The power and flexibility of KNOMAD is in its layered architecture and in the power and flexibility of the various tools. The rule management system is the only tool that has been implemented so far. The following section describes this tool in detail and makes apparent the power of the rule language and flexibility of inference that the rule system has.

3.3.1 The Rule Management System

The rule management system consists of two layers. The base layer is the rule language. This language defines what rules are and how they are evaluated and interpreted. The second layer defines a knowledge base management system (KBMS) that is built on top of the rule language. While the KBMS is a layer on top of the rule language, it does not add anything to it; it is completely defined in terms of the rule

language⁴. The correct way to think of this is that the rule language is the basic level of expression for the knowledge engineer. The knowledge engineer then writes a number of rules that define the KBMS. This KBMS is like another shell that may be instantiated with a domain application, thereby defining an expert system.

In KNOMAD, the rule language and the KBMS work hand in hand. The KBMS is a predefined object available to the knowledge engineer. However, the KBMS definition may be easily modified and extended for special applications. Furthermore, control strategies and conflict resolution strategies may be defined using the rule language itself or by writing procedures (a compiled form of the strategy). These user definable strategies in combination with the expressibility of the rule language provide the power to the rule management system for application to a wide variety of domains.

The KBMS defines a knowledge base to consist of a number of rule groups as well as domain knowledge. Each rule group provides mechanisms for defining aspects of the inference strategy using either further rule groups or user-defined functions. A rule group also has a mechanism for specifying the level of determinism desired over rule execution ([3, 11]). A mechanism for explicitly stating a level of deterministic control over a set of rules is one of the things missing from many expert system shells. Without this, the only way to get the necessary control is by inserting control knowledge into the rules themselves—where it does not belong. By explicitly stating the difference between control over knowledge and the knowledge itself, the maintenance problem becomes easier.

The way control is added is by examining the inference cycle of an expert system. Basically, the rule group inference strategy consists of a match, evaluate, and fire cycle. The match phase is supported by the database interface which automatically queues up potential rules on the tickled-queues of the rule group. The evaluate phase then checks the rules on the tickled-queues to determine which ones belong in the conflict set. Finally, one rule is selected and fired. This approach is completely non-deterministic in selecting which rules get fired from the conflict set. In between the match and evaluate phases a control phase is added. This control phase consists of the definition of a regular expression that defines which rules may be examined and possibly fired next. In this implementation the transition table derived from the regular expression is used instead of the regular expression itself. This is much simpler from the user's perspective. For the user defining a number of rules and wanting to insert some control over them, it is easier to specify a transition table over the rules than it is to define the regular expression that the transition table can be derived from. Furthermore, it is easier to maintain a transition table during rule group modification.

The language of the rule management system is very complex. It supports more traditional data access such as facts and frames. It supports procedural execution directly from rules in the form of messages. This is how the KBMS is initially linked to the rule language. A default execution strategy, including control strategy and conflict resolution strategy exists which may be started from a rule using the message form. The rule language also supports a form of typed quantification.

3.3.2 Tractability and the Rule Management System

Considering the generality and expressivity of the rule language and the KBMS, efficiency has the potential of being forgotten, both in development and implementation. There are three ways of maintaining tractability in the rule language.

The first way of maintaining tractability is to manage the match phase of the inference cycle efficiently. The conceptual definition of the match phase is to check each rule and evaluate its LHS and if it is satisfied then to queue the rule for later interpretation. Obviously, this is also the slowest approach. The step this implementation takes is to compile rules into a rule constraint network that maintains rules in a form more suited for recognizing when data becomes available that has the potential of satisfying the LHS of a rule. To be specific, all the reference variables of the LHS of a rule must have values in order to determine

⁴ However, to make this work efficiently there has been some optimization. The rule language makes certain assumptions about the existence of certain KBMS domain constructs. An example of this is the assumed existence of a rule-group frame with a "rules" slot and a "lhs-tickled-queue" slot.

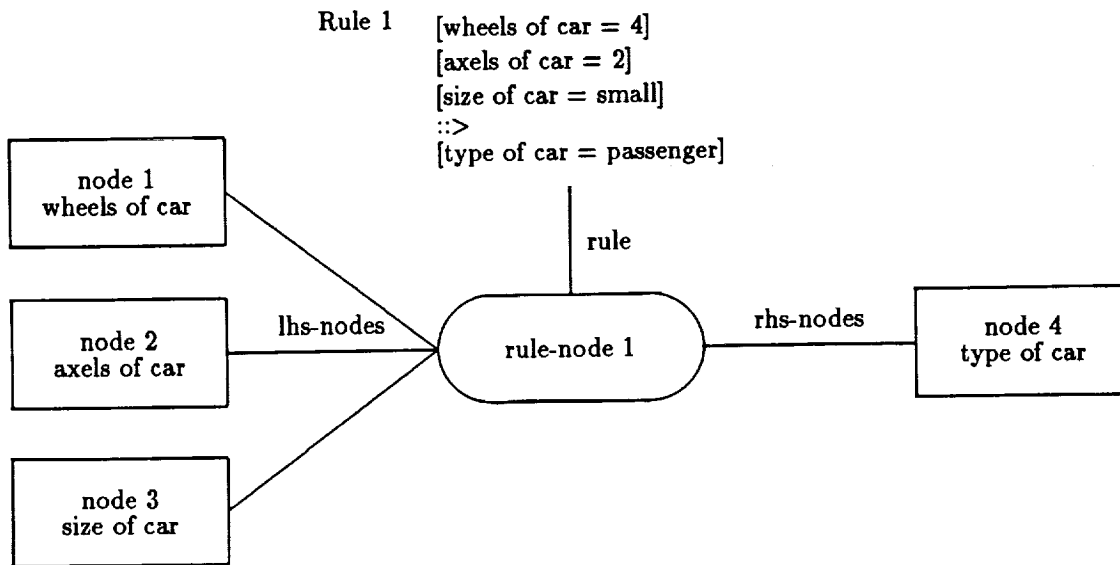


Figure 2: Rule Constraint Network Example

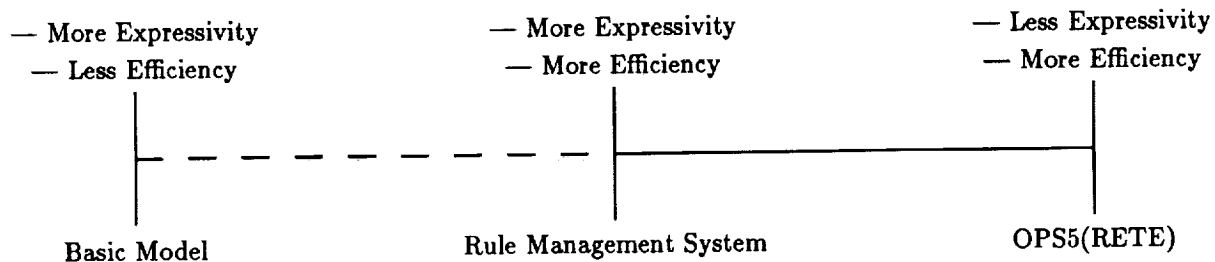


Figure 3: Expressivity versus Efficiency

if the LHS is satisfied. The rule constraint network represents variables as nodes in the network and rules as rule-nodes (see Figure 2). As variables get values, the nodes representing the variables are activated. All the rule-nodes connected to the node are then activated. These rule-nodes are then checked to see if all the nodes representing the LHS variables have values. If so, the rule represented by the rule-node is then put on the appropriate activation queue. As stated, this method allows rules to be queued that may not be satisfied. The rule constraint network only checks to see if variables have values, not if they have the correct value. Thus the rules on the tickled queues must still be evaluated for satisfaction.

Alternatively, rules could be compiled more completely so that variables are checked to see if they have the right value. The complexity of the rule language implemented here makes this a difficult task and it has not been determined that it would be cost effective. Languages with less expressivity can be completely compiled much more easily (such as OPS5 using the RETE network [9] [10]). Expressivity and tractability is always lurking behind the scenes; see Figure 3 for an idea of where the rule management system fits.

The second way of maintaining tractability is by providing code for inference strategies instead of providing declarative rules defining the inference strategy. This is making use of the compiled versus interpreted option. This means that a programmer fluent in the rule language and its implementation needs to be available for the knowledge engineering task. However, considering the expressivity of the rule language, this may be a cost effective option. It is very easy for the programmer to express what he wants without having to go into contortions over representational limits.

The third way to maintain tractability is by the effective use of knowledge. This is to make use of Raj Reddy's fifth principle: *Knowledge eliminates the need for search* ([24]). In other words, the domain to be represented needs to be analyzed for maximum efficiency in terms of knowledge organization. For example, I can have two rule groups; one rule group forward chains on various data and computes a value for the variable *diagnosis*. The other rule group then uses the value of *diagnosis* to output the results to the user. If there are 50 rules in each rule group that use the variable, then 100 rules are triggered whenever the value of the variable changes. If both of these rule groups are made to be sub-rule groups of a control rule group, the first rule group can compute a value for *diagnosis*. The control rule group can then set the value for the variable *the-diagnosis* as the value of *diagnosis*. *the-diagnosis* is then used by the second rule group instead of *diagnosis*. Now only 50 rules get triggered at any time.

The representation of knowledge should make maximum use of divide and conquer principles of knowledge organization. Another approach along the same lines is to provide strong heuristic knowledge to eliminate the need for search. This can come in many forms including domain guided inference strategies over rule groups as well as judicious use of control over the execution of rules in a rule group (i.e., the transition table method of control over rules).

3.4 Mixins

Diverse knowledge representation requirements may be needed by various domains. These may include temporal knowledge and non-monotonic knowledge. Neither of these have been implemented in KNOMAD and may involve major modification to the database interface and structure. Conceptually, the use of these aspects of knowledge are optional depending on the needs of the application. This is why they are considered as a mixable item to those items that are necessary for a collection of tools.

Temporal knowledge, non-monotonic knowledge, and uncertain knowledge should be a set of extensions to the database and database interface. They have not been implemented and therefore will not be discussed in any more detail.

4 Operational Scenario

In this section the operation of the rule management system of KNOMAD is described further by way of an example.

The Freight Agency Knowledge Base is a toy knowledge base consisting of a ten rule rule group that determines how to send a package given its length, width, and height. The rule group also needs to know if the package needs to be sent urgently as well as the destination country.

4.1 The Knowledge Base

A knowledge base for a knowledge agent consists of the following syntactic parts.

```
KB : <kb-name>
Domain : <domain file-name>
Rule-Group : <rg-name>
    <rg contents>
Rule-Group*
Domain-Knowledge
Begin : <starting rule groups>
End-KB
```

First the knowledge base is identified. Domain knowledge for the knowledge base can be given in two ways. Usually the model of the domain is defined in some separate file and identified next. After the domain knowledge the rule groups that make up the knowledge base are given. Then further domain knowledge may be defined. This set of domain knowledge is generally used for domains requiring very little domain knowledge as well as domain knowledge that is specific to the syntax of the rules in the

knowledge base. A begin statement follows next. The begin statement allows one or more of the given rule groups to be executed in parallel with one another. A Hearsay application would probably execute all its rule groups in parallel with one another, for example. Finally an end statement is given.

The knowledge base for the Freight Agency Knowledge Base is as follows.

```

@@
@@ The Freight Agency Knowledge Base
@@
@@ This knowledge base defines two rule groups, the main one for
@@ determining how to send a package, and a secondary one for implementing
@@ the control strategy of the first.

KB : freight

@@
@@ This lets us physically put the rule groups in separate files.
@@
FILE : ~/knowad/kbms/freight-rules.rg @ load the freight rules
FILE : ~/knowad/kbms/freight-cs.rg @ load a control strategy rule group

@@
@@ The domain knowledge. A number of constants and facts particular
@@ to this knowledge base.
@@
Domain-Knowledge : @ these are constants just to this kb
constants :
    europe ;
    america ;
    asia ;
    australasia ;
    road ;
    rail ;
    special ;
    sea ;
    air ;
    no ;
    yes ;
    true ;
    :ok ; :ng ; :missing-patterns .
facts : @ some initialized facts
    european-countries =
        ( france belgium spain germany uk portugal italy austria poland ) ;
    american-countries = ( usa canada mexico brazil ) ;
    asian-countries = ( china japan india ussr ) ;
    australasian-countries = ( australia new-zealand ) ;
    modes = ( road rail special sea air ) ;
    empty = ( ) .

@@
@@ Initially we want to begin execution of the main rule group: rg1.
@@
Begin : RG1

@@
@@ That's All Folks
@@
End-KB

```

4.2 The Rule Group

A rule group consists of three parts. The first part is header information used for defining how the rule group is to be executed. The second part is the rules themselves. The final part is a termination

condition that defines when the rule group is done.

There are three items that may optionally be specified in the header information for a rule group. The first allows a user to define a transition table that the rules will follow. As an example, the transition table for the main rule group of the Freight Agency Knowledge Base is as follows.

```
CONTROL : ((start (frule10))
  (frule10 (frule1 frule2 frule3 frule4 frule5 frule6 frule7
    frule8 frule9))
  (frule1 (frule1 frule2 frule3 frule4 frule5 frule6 frule7
    frule8 frule9))
  (frule2 (frule1 frule2 frule3 frule4 frule5 frule6 frule7
    frule8 frule9))
  (frule3 (frule1 frule2 frule3 frule4 frule5 frule6 frule7
    frule8 frule9))
  (frule4 (frule1 frule2 frule3 frule4 frule5 frule6 frule7
    frule8 frule9))
  (frule5 (frule1 frule2 frule3 frule4 frule5 frule6 frule7
    frule8 frule9))
  (frule6 (frule1 frule2 frule3 frule4 frule5 frule6 frule7
    frule8 frule9))
  (frule7 (frule1 frule2 frule3 frule4 frule5 frule6 frule7
    frule8 frule9))
  (frule8 (frule1 frule2 frule3 frule4 frule5 frule6 frule7
    frule8 frule9))
  (frule9 (frule1 frule2 frule3 frule4 frule5 frule6 frule7
    frule8 frule9)))
```

While this looks inordinately complex for a ten rule rule group, the following shows the transition table for an 89 rule rule group that is used for diagnosing faults in a power system.

```
CONTROL :
  ((start (init-rule))
    (init-rule (rule1))
    (rule1 (rule2 rule2a))
    (rule2 (rule3 rule4.1 rule5 rule31 rule31.1 rule31.2 rule35
      rule35.1))
    (rule2a (rule3 rule4.1 rule5 rule31 rule31.1 rule31.2 rule35
      rule35.1))
    (rule3 (rule4 rule4a))
    (rule4 (rule20))
    (rule4a (rule32))
    (rule32 (rule33))
    (rule33 (rule34))
    (rule4.1 (rule4.2))
    (rule4.2 (rule4.3))
    (rule4.3 (rule4.4 rule4.5 rule4.6 rule4.7 rule4.8))
    (rule4.8 (rule4.9 rule4.10 rule4.11 rule4.12 rule4.13
      rule4.14 rule4.15 rule4.16 rule4.17))
    (rule5 (rule6 rule6.1))
    (rule6 (rule7 rule8 rule9))
    (rule9 (rule10 rule11 rule12 rule13 rule15 rule17 rule18))
    (rule6.1 (rule19))
    (rule35 (rule36 rule37 rule38 rule39 rule40 rule41 rule42 rule43))
    (rule35.1 (rule44))
    (rule44 (rule45 rule46 rule47))
    (rule47 (rule48 rule49 rule50 rule50.1 rule51 rule52
      rule53 rule54 rule55 rule56))
    (rule56 (rule57 rule58))
    (rule58 (rule59 rule60))
    (rule60 (rule61))
    (rule61 (rule62 rule63 rule64 rule65 rule66))
    (rule66 (rule67))
    (rule67 (rule68 rule69 rule70 rule71 rule72 rule73 rule74)))
```



```
(rule74 (rule75))
(rule75 (rule77))
(rule77 (rule56)))
```

These examples show that there is quite a bit of latitude in specifying the amount of determinism one wants in a rule group. The second example includes an iteration pattern between rules 56 and 77. Even though the second example defines the transition table for 89 rules, not all 89 rules are explicitly represented. This is because a large number of them conclude a pattern that will allow the termination condition to succeed, indicating that the rule group is done.

The control strategy of a rule group may also be defined in the header of a rule group. This is done by specifying the name of either a function or of another rule group. If it is the name of another rule group, that rule group will be used as the control strategy. This is what has been done for the Freight Agency Knowledge Base.

Finally, the conflict resolution strategy of a rule group may be specified as the control strategy is specified.

The main body of the rule group consists of the rules. The third section of the rule group simply consists of an optional single condition, that if true, indicates the successful completion of the rule group. If a termination condition for the rule group is not given, the rule group is never finished.

The rule group (minus transition table) for the Freight Agency Knowledge Base follows.

```
%%
%%      @($)freight-rules.rg 1.2      1/31/90
%%
%% This rule group determines how to send a package.
%% It is assumed that the package
%% is being sent from somewhere in europe. Initially, we need to compute the
%% volume of the package based on its length, width, and height. This is done
%% in frule10. The zone of where the package is sent is computed in frules6-9
%% based on the country being shipped to. The only other piece of information
%% that might be needed is whether or not it is urgent.
%%
rule-Group : RG1
%%
%% The control strategy for this rule group is specified to be declared as the
%% rule group freight-cs.
%%
CONTROL-STRATEGY : freight-cs
%%
%% Rules are read as IF <condition> THEN <condition>
%% where the actual syntax is <condition> ::> <condition>
%%
%% Frule1 (in english):
%% IF the zone is europe AND the distance is less than 100
%%    AND the volume is less than 100
%% THEN the mode to send the package is road.
%%
Frule1
[ zone = europe ]
[ distance < 100 ]
[ volume < 100 ]
::>
[ mode = road ]
;
%%
%% Frule2:
%% IF the zone is europe AND the volume is less than 200
%%    OR
%%    the distance is greater than or equal to 100
%%    AND the volume is less than 100
%% THEN the mode is rail
%%
```

```

Frule2
[ zone = europe ]
[ volume < 200 ]
OR
[ distance >= 100 ]
[ volume < 100 ]
::>
[ mode = rail ]
;
Frule3
[ zone = europe ]
[ distance >= 100 ]
::>
[ mode = special ]
;
Frule4
[ zone <> europe ]
OR
[ urgent = no ]
[ volume >= 50 ]
::>
[ mode = sea ]
;
Frule5
[ zone <> europe ]
[ urgent = yes ]
[ volume < 50 ]
::>
[ mode = air ]
;
Frule6
[ country memberin european-countries ]
::>
[ zone = europe ]
;
Frule7
[ country memberin american-countries ]
::>
[ zone = america ]
;
Frule8
[ country memberin asian-countries ]
::>
[ zone = asia ]
;
Frule9
[ country memberin australasian-countries ]
::>
[ zone = australasia ]
;
Frule10
::>
[ volume = length TIMES width TIMES height ]
;
[ mode memberin modes ]

```

4.3 The Execution of a Knowledge Agent

The execution of a knowledge agent reduces to executing the rule groups specified in the begin statement of the knowledge base. The execution of a rule group consists of initialization followed by a loop that terminates when a rule group's termination condition is satisfied. The loop a rule group executes consists of five steps: Step one is to check if the termination condition is satisfied, step two determines which

rules are viable based upon the transition table of the rule group, step three runs the control strategy for the rule group and computes the conflict set, step four runs the conflict resolution strategy of the rule group and computes the fire set, step five then fires the rules in the fire set.

Since the rule management system uses a data-driven approach, if no rules have been queued up that may potentially be fired, it is pointless to busy-wait through this loop. Therefore, the process executing the rule group is woken up when there is work to do.

This model defines a very simple, yet very flexible approach to defining and executing knowledge agents. In the Freight Agency Knowledge Base, the rule group waits until data is available to fire a rule on. According to the transition table given earlier it will first try to fire rule10. That rule determines the volume of the package to be sent based on its length, width, and height. Once that rule fires, the remaining rules fire nondeterministically until a value is computed that determines the mode with which to send the package.

For brevity the control strategy rule group of the Freight Agency Knowledge Base is not given here.

5 Discussion

The two most controversial aspects of the rule management system are probably the transition table method of control and the ability to specify rule groups as part of the execution strategy of a rule group.

There are two views to control in a rule group. One view prefers to embed control knowledge about how one rule fires after another rule in the rules themselves. The other view prefers to make this explicit by specification in terms of a transition table (or regular expression). Both views have their strong points. KNOMAD takes the second view. It is the position of this author that it is easier and more comprehensible to keep this knowledge explicit. It also forces the knowledge engineer to maintain an accurate representation of the rules in the rule group. If a rule group starts becoming much too complex to understand, it is probably an indication that it may be wise to split the rule group into sub-rule groups or rethink what the rule group is doing. However, KNOMAD and the rule management system don't *know* what a rule states, therefore it is still possible for a person to take the first view while using KNOMAD.

The other controversial aspect is the ability to use one rule group to control another rule group. This isn't so much a controversial part of KNOMAD as it is an *inefficient* part. That is, inference is slowed down an order of magnitude if this method is used. However, the point is that this provides the ability to investigate new flavors of execution of a rule group before going down to LISP code and implementing it efficiently. It is certainly not the intention of this paper to propose that these meta-rule groups be used in a production system where speed is an issue.

The most challenging part of the development of KNOMAD and the rule management system has been defining semantics. It is really quite amazing how complex an issue this really is. An OPS5 type system has a relatively simple semantics. As rules are made more complex—disjunction is provided on LHS and RHS, else conditions are added, quantification is provided at various levels—the meaning of executing a rule takes on a temporal significance. When this complexity is available, it is not enough to take a static view of working memory to determine how to fire a rule. Similar to Prolog, a theory of the world must be maintained and kept consistent. In the rule management system presented here, most of this complexity has been eliminated in favor of completing an initial system that is well-defined and relatively efficient.

6 Conclusions and Future Work

The architecture of KNOMAD has been shown to be both flexible and powerful. Three very different knowledge agents have been defined so far. The Freight Agency Knowledge Base is the simplest. However, it was made more complex by using a meta-rule group for its control strategy. The monkeys and bananas problem has been defined. It consists of about twenty rules and uses quantification extensively. It has

been very useful to drive out limitations and inadequacies of the rule system. Finally, a large knowledge base consisting of three rule groups and a total of about 160 rules has been defined that performs fault diagnosis in a power management and distribution system for a space station like platform ([26]). In addition to the rules for this knowledge agent, the domain requires over 1500 facts to describe the model. This last application has proved very successful.

6.1 Future Work

There are a number of issues that just this part of KNOMAD's implementation raises. The rule management system could be made more efficient by examining the applicability of RETE-net structures to rules. There are cases where partial matches are appropriate and RETE structures would be too much. However there are also cases where the RETE structures would be very relevant. The RETE structures would have to be done in such a manner that they didn't enforce a particular direction on the rule (or perhaps enforced both forward and backward chaining).

Another application for future work is to use databases in an integrative manner. Right now a database is used as working memory and provides communication between knowledge agents as well as object persistence. It would be useful to use data in existing databases as well as make data that is the result of inference available to other applications. This requires a substantial amount of research to determine the effects of temporal inferences as well as the more traditional problems of locking and synchronization.

Adding planning and scheduling to KNOMAD is the next step in further development. This requires a look into reasoning about temporal data and constraint systems and adding these as well. It may also be necessary to determine how to deal with nonmonotonic data as we look at temporal reasoning as well as more complex rules (e.g. RHS disjunction in a forward chaining system).

7 Acknowledgements

This work was performed by Martin Marietta Astronautics Group under contract number NAS8-36433 to NASA, George C. Marshall Space Flight Center, Huntsville, Alabama.

References

- [1] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832-843, November 1983.
- [2] Barry R. Ashworth. Reactive autonomous planning in spacecraft. In *Proceedings of the Conference on Aerospace Applications of Artificial Intelligence*, 1989.
- [3] A. B. Baskin. Combining deterministic and non-deterministic rule scheduling in an expert system. In *AAMSI*, 1986.
- [4] Bruce G. Buchanan and Richard O. Duda. *Principles of Rule-Based Expert Systems*. Technical Report HPP-82-14, Stanford Heuristic Programming Project, 1982.
- [5] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4), 1989.
- [6] David Chapman. *Planning for Conjunctive Goals*. MIT Industrial Liaison Program Report 10-20-86, Massachusetts Institute of Technology, 1986.
- [7] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24(3):347-410, 1984.

- [8] K. Forbus. Qualitative process theory. *Artificial Intelligence*, (24), 1984.
- [9] Charles L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1), 1982.
- [10] Charles L. Forgy and Susan J. Shepard. Rete: a fast match algorithm. *AI Expert*, January 1987.
- [11] M.P. Georgeff. Procedural control in production systems. *Artificial Intelligence*, 18:175-201, 1982.
- [12] D. Haussler. Bias, version spaces, and valiant's learning framework. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 324-336, 1987.
- [13] P.J. Hayes. The logic of frames. In B.L. Webber and Nils J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 451-458, Morgan Kaufmann, 1981.
- [14] Frederick Hayes-Roth. Towards benchmarks for knowledge systems and their implications for data engineering. *IEEE Transactions on Knowledge and Data Engineering*, 1(1), 1989.
- [15] Guy Lewis Steele Jr. and Gerald Jay Sussman. *Constraints*. Technical Report AI Memo No. 502, Massachusetts Institute of Technology Artificial Intelligence Laboratory, November 1978.
- [16] J.E. Laird, A. Newell, and P.S. Rosenbloom. Soar: an architecture for general intelligence. *Artificial Intelligence*, 33, 1987.
- [17] Doug Lenat and R.V. Guha. *The World According to CYC*. Technical Report ACA-AI-300-88, Microelectronics and Computer Technology Corporation, September 1988.
- [18] Douglas B. Lenat. Ontological versus knowledge engineering. *IEEE Transactions on Knowledge and Data Engineering*, 1(1):84-88, March 1989.
- [19] Hector J. Levesque. Knowledge representation and reasoning. *Annual Reviews of Computer Science*, 1986.
- [20] C.L. Liu. *Elements of Discrete Mathematics*. McGraw Hill Book Co., 1985.
- [21] Satoshi Matsuoka and Satoru Kawai. Using tuple space communication in distributed object-oriented languages. In *OOPSLA*, 1988.
- [22] M. Minsky. A framework for representing knowledge. In P. Winston, editor, *The Psychology of Computer Vision*, pages 211-277, McGraw-Hill, 1975.
- [23] A. Newell, J.C. Shaw, and H.A. Simon. A general problem-solving program for a computer. In *Information Processing: Proceedings of the International Conference on Information Processing*, pages 256-264, UNESCO, Paris, 1960.
- [24] Raj Reddy. Presidential Address at the American Association for Artificial Intelligence Conference, 1988.
- [25] R. Reiter. Towards a logical reconstruction of relational database theory. In M. Brodie, J. Mylopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling*, Springer-Verlag, 1984.
- [26] Joel D. Riedesel, Chris Myers, and Barry Ashworth. Intelligent space power automation. In *Proceedings of the Fourth IEEE International Symposium on Intelligent Control*, 1989.
- [27] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23-41, January 1965.
- [28] Y. Shoham. *Reasoning about Change*. MIT Press, 1987.

- [29] Mark Stefik, Jan Aikins, Robert Balzer, John Benoit, Lawrence Birnbaum, Frederick Hayes-Roth, and Earl Sacerdoti. The organization of expert systems. *Artificial Intelligence*, 18, 1982.
- [30] Mark Stefik and Daniel G. Bobrow. Object-oriented programming: themes and variations. *The AI Magazine*, 40-62, 1986.
- [31] P. Thyagarajan and Arthur M. Farley. *Design and Implementation of a Qualitative Constraint Satisfaction System*. Technical Report CIS-TR-87-03, Department of Computer and Information Science University of Oregon, March 1987.
- [32] Steven A. Vere. Planning in time: windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(3):246-267, May 1983.
- [33] Masanobu Watanabe, Toru Yamanouchi, Masahiko Iwamoto, and Yuriko Ushioda. Cl: a flexible and efficient tool for constructing knowledge-based expert systems. *IEEE Expert*, 41-50, Fall 1989.
- [34] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, 1988.

**A KNOWLEDGE BASE ARCHITECTURE
FOR DISTRIBUTED KNOWLEDGE AGENTS**

**A PAPER PRESENTED AT THE AI FOR SPACE APPLICATIONS CONFERENCE
IN JUNE 1990.**

A KNOWLEDGE BASE ARCHITECTURE FOR DISTRIBUTED KNOWLEDGE AGENTS

Joel Riedesel
MS: S-0550
Martin Marietta Astronautics
P.O. Box 179
Denver, Co. 80201
jriedesel@den.mmc.com

Bryan Walls
NASA, Marshall Space Flight Center
Bldg. 4487 EB12
Huntsville, AL 35812

ABSTRACT

In this paper a tuple space based object oriented model for knowledge base representation and interpretation is presented. An architecture for managing distributed knowledge agents is then implemented within the model.

The general model is based upon a database implementation of a tuple space. Objects are then defined as an additional layer upon the data base. The tuple space may or may not be distributed depending upon the database implementation. A language for representing knowledge and inference strategy is defined whose implementation takes advantage of the tuple space. The general model may then be instantiated in many different forms, each of which may be a distinct knowledge agent. Knowledge agents may communicate using tuple space mechanisms as in the LINDA model as well as using more well known message passing mechanisms.

An implementation of the model is presented describing strategies used to keep inference tractable without giving up expressivity. An example applied to a power management and distribution network for Space Station Freedom is given.

1. Introduction

In this paper a tuple space based object oriented model for knowledge base representation and interpretation is presented. The model provides a general knowledge language that is at once expressive and extendable. This allows it to be applied to many different domains including knowledge base management systems for expert system shells and architectures for distributed knowledge agents.

The field of Distributed Artificial Intelligence (DAI) is very complex. Besides the problems involving representing any particular agent, there is a whole new set of problems that are concerned with how multiple agents communicate with one another. This problem is more than just defining a mechanism but also involves protocols. How does one model of communication enhance the ease of solving one problem over another model?

The model presented in this paper supports DAI at a low level. This model presents a framework for defining multiple knowledge agents that must coordinate and cooperate with one another to solve some problem. This is different than most papers about DAI in that most papers are concerned with problems of communication and cooperation protocols. This model defines an architecture that supports the definition and implementation of diverse knowledge agents and their necessary protocols as the problem requires.

1.1. Requirements for Representing Distributed Knowledge Agents

There are a number of requirements a model will need to represent distributed knowledge agents adequately. Four interrelated requirements are identified and discussed here: Domain independence and expressivity, control knowledge, and communication.

1.1.1. Domain Independence and Expressivity

Domain independence and expressivity of a model are very closely related. Domain independence is concerned with the ability of the model to represent problems from any domain. The particular representation and storage of knowledge is important to domain independence. However, more than simply being concerned with expressing a problem in the model, domain independence is concerned with the ability of the model to integrate with the various problem domain environments. Expressiveness is concerned particularly with the ability and ease of stating a problem in the model language and not how the problem might have to deal with the environment of the problem.

The requirement for domain independence is concerned with the ability to represent a problem in the model and integrate that problem solution into the problem environment. This really implies that the language of the model must be extensible. It must be possible to enlarge the language using the base language as a start. This includes the ability to change inference mechanisms and define new models of inference. While most languages are concerned with representing data, this language is also concerned with representing control knowledge.

To support this requirement the language of the model presented here is a rule language built using object-oriented programming and extendable using the objects of the language. Rules are a primitive object and are evaluated and interpreted based upon a generic view of data stored in an independent database. A basic rule group object is provided with a forward chaining inference mechanism. A rule group may also be used to control the execution of another rule group thus enabling the definition of new inference mechanisms using the language. Additionally, more specific rule groups may be defined as sub-classes of the base rule group to support different inference mechanisms defined with meta-rule groups.

The other hand of domain independence is expressivity. There are at least two aspects to expressiveness: domain knowledge representation and control knowledge representation.

The problem of expressivity is that as more expressivity is allowed, along with more domain independence, the more intractable a language may become. The basic inference model for knowledge based systems consists of a match-select-fire cycle. The match phase determines those rules which are enabled and may be fired. The select phase selects one rule from the matched rules and the fire phase fires, or interprets, the selected rule. This basic paradigm captures the model of inference that knowledge base systems perform. To make this efficient, there are a number of options to the knowledge base designer. The language may be restricted, for example providing only universal quantification. Various compilation mechanisms may also be incorporated to make the match phase as static as possible. Both of these mechanisms are performed in OPS5 using the RETE network ([7, 8]).

The model presented here provides a large amount of expressivity while moving from the basic inference model presented above to something more tractable such as OPS5 and the RETE network.

To provide a large amount of expressiveness, the language of the model provides a frame system for representing structured knowledge as well as simple facts. Rules may use data from frame knowledge and fact knowledge. In [11], Hayes-Roth presents a number of knowledge categories that are needed for benchmarking different knowledge base systems. The categories fact, rule, class, entity, relation, and structure are provided by the model presented here. The remaining categories are not provided for but are planned as future work.

1.1.2. Control Knowledge Representation

Another requirement is the representation of control knowledge. The interpretation of the knowledge of different knowledge agents will need to be based upon the needs of the different knowledge agents. Some may require forward chaining while some may require backward chaining. Some may require more exotic strategies such as forward chaining with beam search.

There are two aspects of control knowledge identified here. One is the inference strategy used over a rule group and may be provided by defining meta-rule groups or LISP code. The other way of controlling rules is by providing a level of determinism into the ordering of rules themselves [1, 9]. The way this is done in most expert system shells is by adding variable references into rules that provide control (e.g. IF [step = 1] ... THEN ...). This makes maintenance of the rules very difficult. An alternative is to use a transition table, allowing any level of non-determinism. Each entry in the transition table is used to index into the next possible rules. This is equivalent to defining a regular expression over a rule group ([9]).

The mechanism provided here is a transition table. As a rule is fired, that rule (its name) is used to index into the transition table to determine what rules are allowed to be fired on the next cycle. If the particular inference strategy being used allows more than one rule to be fired in a particular cycle, each of the fired rules is used to index into the transition table and the resulting lookups are unioned together. Complete non-determinism may be provided by not using a transition table and complete determinism may be provided by specifying only one rule as the next rule for any particular rule fire. Any mix of determinism and non-determinism may be specified using this mechanism. This is discussed further in section 2.

1.1.3. Robust Communication

The last requirement is for mechanisms for communication and coordination between knowledge agents. There is definitely not a consensus on the best mechanism for communication in the literature. The basic mechanism for communication and coordination provided here is the tuple space model ([5, 14]).

The tuple space model allows the addition of knowledge agents without having to modify existing knowledge agents about the communication interactions of the new agents. Communication is performed by inserting and removing tuples from the tuple space. If a knowledge agent is defined to take action based upon the existence of a tuple, the data-driven nature of the database will notify the knowledge agent of the ability of the rule to fire.

In addition to the tuple space model, it is entirely possible to perform message passing and other forms of communication by defining new functions that implement message passing to the knowledge language. Thus the tuple space, although probably the primary mechanism for communication in this model, need not be at all restrictive to robust communications.

1.2. The Model Overview

Figure 1 shows the general architecture for supporting distributed knowledge agents. The database, to support the common tuple space, is the only module common to the agents. Even then, a common database is not a requirement if alternative forms of message passing are chosen. The database interface and Knowledge Base Management System (KBMS) are instantiated once on each physical platform.

In the figure a database is shown that exists independently of the KBMS. How the database is implemented is not important to the operation of the KBMS. The database interface is concerned with interfacing knowledge agents to the database. If a tuple is asserted to the database by a knowledge agent, the database interface will both add it to the database as well as notify any other knowledge agents that use that tuple of it. Rule Group 1 represents one knowledge agent and Rule Group 2 represents another. Both of these knowledge agents happen to be defined in a single instantiation of the KBMS system. Rule Group 3 and 4 are sub-rule groups and are part of the

knowledge agent consisting of Rule Group 1. Another way to think of it is that each rule group represents a distinct knowledge agent where Rule Group 3 and 4 are strictly controlled by the agent of Rule Group 1. If another knowledge agent existed on another physical platform, there would be an additional instantiation of a database interface and KBMS. The database itself would remain (whether distributed or not) the same.

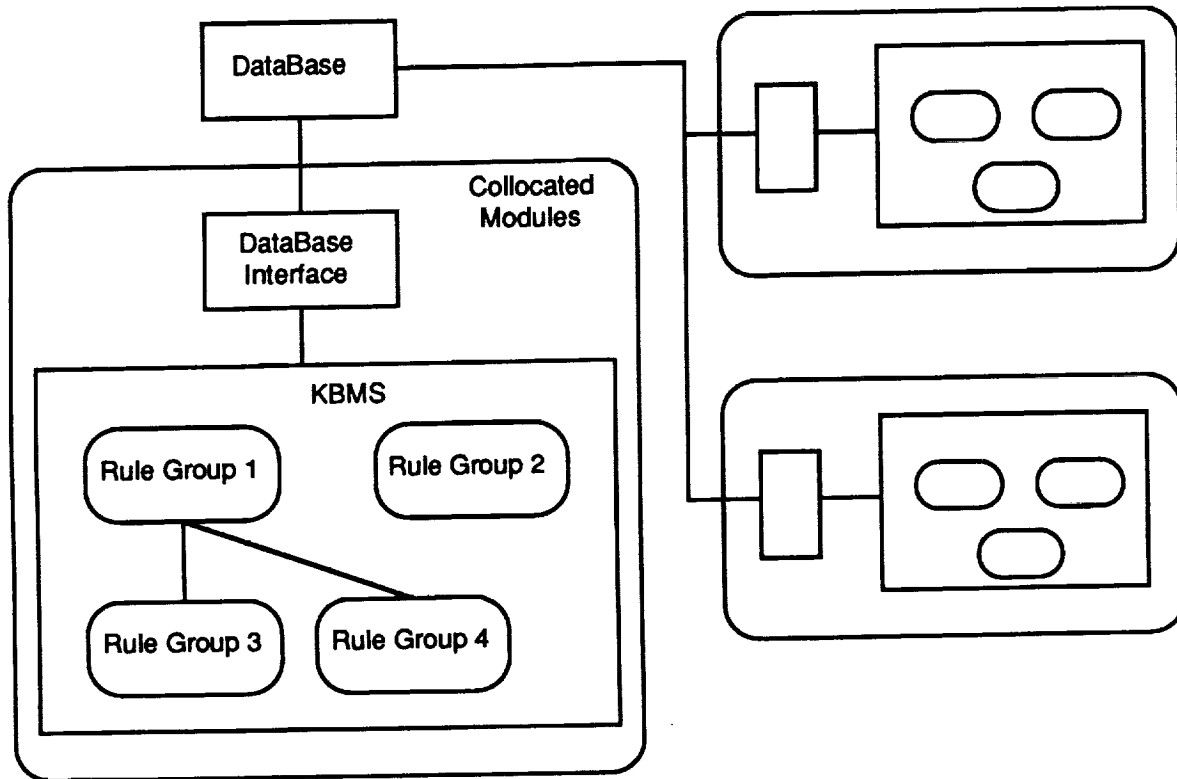


Figure 1 - KBMS Architecture

2. The Model: Its Design and Implementation

The model described here is based on a shared database, used to communicate between various knowledge agents through the passing of tuple data structures. Frames are defined on this base through a database interface to the knowledge agents, allowing data abstraction, inheritance, and structuring. Each knowledge agent contains domain knowledge in the form of rules and data and is controlled by the KBMS. The knowledge language of the KBMS provides the support to implement knowledge agents and allows arbitrary function calls. Tractability is maintained by a combination of rule compilation provided by the system and heuristic control knowledge provided by the knowledge engineer. The following subsections describe the design of the KBMS in much more detail.

2.1. The Base: The Database and Tuple Space

The first level of support for a KBMS is working memory, or the database. The architecture presented here views the database as a plug compatible module. This provides the ability to take advantage of existing databases and database management mechanisms such as ORACLE or INGRES for example.

The KBMS takes a tuple space view of the world and requires that the database represent them. A tuple is an ordered sequence of (possibly) typed fields. Conceptually, a tuple may be

thought of as an object that exists independent of the process that created it. This implies that a place to store it is needed if it is to live a life independent of processes used to create and destroy it. The tuple space is sufficient for representing First Order Logic (FOL) and also sufficient for representing at least relational databases ([17]).

The tuple space has also been proposed as a mechanism for supporting distributed processing (the LINDA model, see [5]). There are three basic mechanisms that a process may perform on tuples in the tuple space: IN, OUT, and READ. The tuple fields in the IN and READ operations may optionally contain variable arguments for matching. IN and READ block until the tuple is present in the database. IN subsequently removes the tuple from the database while READ does not. OUT installs a tuple in the database. The benefit to distributed processing that the tuple space provides is the ability to add processes to the environment and have them communicate with existing processes without first having to encode knowledge about existing processes directly into their formalism. It has also been proposed that these basic tuple space operations are sufficient for supporting other communication mechanisms such as those defined in contract nets and Actors. On the other hand, the use of IN must be judicious in order to support the addition of other processes that may also need to use the tuple. See [5, 14] for more details.

The database provides three operations similar to the tuple space operations, these are: STORE, RETRIEVE/MATCH, and DB-REMOVE. The major difference between these operators and the tuple space operators IN, READ, and OUT, is that these do not block. Another difference is that while IN and READ will non-deterministically select one of multiple matching tuples, MATCH will return all of them (RETRIEVE will also act non-deterministically). Although the database operations defined here are not identical to the tuple space operations in the LINDA model (they do not block), the operations, in combination with the data-driven nature of a KBMS are probably sufficient for managing distributed processing protocols as LINDA does ([5]).

This view of the database allows for many different implementations of the database to be used, including distributed databases. A distributed database would provide the ability to support distributed knowledge agents defined using the knowledge representation presented here. Their communication is then supported by the tuple space mechanism as implemented in the distributed database.

The implementation of the database for the KBMS described here also supports a form of integrity constraints and a restricted version of views. An integrity constraint is defined by a tuple. Fields of the tuple may contain one of four items: An actual field descriptor (formal argument); a variable (to match an argument); a type specifier, meaning that the field may match tuples with the corresponding field of the same type; and finally, a type declarator, meaning that for all tuples matched on the other three options, this field must be of the declared type. Views provide different databases for storage and retrieval. This is quite different to the traditional use of views in most databases. What this provides is a way to organize data into logical groupings. Currently, the use of integrity constraints is global across all views, in the future this should use the view mechanism just as any other storage or retrieval operation does.

2.2. The Next Level: Data Abstraction and Inheritance

The database interface is the module that links the database to the KBMS and vice versa. The interface provides the necessary hooks for proper accessing and notification of data in the database and of interest to the KBMS. The database interface provides the operations STORE!, REMOVE!, MATCH!, and RETRIEVE!. It also provides a data and procedure abstraction mechanism—a frame system ([10, 15]). Frames are used to extend the knowledge representation language of the KBMS for supporting complex domains requiring novel abstractions and inheritance.

Frames are an abstract organization of data into conceptual units. In this definition data may be any object, it may be simple data or even procedural objects. A frame may have any number of slots. Frames may be defined as children of multiple parents (making inheritance potentially more complex) and may also have code attached to them that is executed whenever a

new instance of the frame or one of its children is created. Slots may have six optional aspects. The most used aspect is the `:value` aspect. This aspect is where a value for the slot is located. The `:if-needed` aspect is used to store code that is executed if a slot value is asked for. The `:if-added` aspect is used to store code that is executed whenever the slot gets a new value. There are two aspects that are used to constrain the value of the slot. The `:constraint` aspect is used to store code that checks if an added value passes the constraint. Since this is user-defined code there is no restriction on what it may do, only that it return a true or false status indicating the result of the constraint. The `:mustbe` aspect constrains the value to be one of a list of formal values or frames. Finally, the `:distribution` aspect is used to determine if the value of the slot is for global distribution, accessible to all knowledge agents, or only for local use.

Frame data is stored in the database as 5-tuples of the form: `(frame <name> <slot> <aspect> <value>)`. Obviously, there is no restriction on the value aspects, they can be any normal data type as well as executable code. Facts are stored in the database as 4-tuples: `(fact <name> :value <value>)`.

Frame inheritance information must also be stored in the database so that inheritance over frames may take place in response to different knowledge agent's requests for data values. The current implementation does not yet store this data in the database.

It is the responsibility of the database interface to both store and retrieve information and to notify the KBMS of changes to data made by other knowledge agents. Thus, if knowledge agent 1 makes a change to the value of a fact and knowledge agent 2 uses that fact on the left hand side (LHS) of a rule, it is the responsibility of the database interface (and distributed database) to notify knowledge agent 2 of the changed fact. And vice versa, it is the responsibility of the system, when knowledge agents are being defined to notify the database interface of which knowledge agents index on which facts.

2.3. The Knowledge Base Management System

Now that the basic knowledge storing and retrieving mechanisms have been outlined, the heart of the system needs to be defined.

The knowledge representation language is defined as a set of objects up to the level of rules. A KBMS is then instantiated in the language using a combination of user-defined objects, user-defined code, rules and KBMS domain knowledge (i.e. knowledge about what the KBMS is). The KBMS instantiated here consists of the user-defined objects: `rule-group` and `knowledge-base`. The default inference strategy is implemented in code, but can also be implemented in the form of meta-rule groups. Rules for defining control strategy, for instance, are defined. The domain knowledge consists of frame knowledge about rule groups, knowledge bases, and knowledge of executable procedures.

To go one level further, an expert system is then instantiated from the KBMS. Here, rules and rule groups are provided for the domain; domain knowledge is provided, as well as more specific inference strategies for the expert system. To use an analogy from the `flavors` object oriented system, the knowledge representation language is like a base flavor. It is necessary to define a mixin to it in order to give it functionality. Finally, the flavor can be instantiated into a user-definable object (e.g. an expert system for fault diagnosis).

The knowledge representation language provides functionality to the level of rules. A rule has the basic form of `LHS ::> RHS`, and may include else conditions. Quantified rules quantify over some set, binding a variable to successive values and executing the sub-rule of the quantified rule. Rules have the basic operations: `evaluate-lhs`, `evaluate-rhs`, `interpret-lhs`, `interpret-rhs`, and `interpret-else`. These operations may return one of three values: `:ok`, `:ng`, and `:missing-patterns`. If the rule evaluates `:ok` is returned, if conditions are not met `:ng` is returned, and finally, if patterns needed to check conditions do not yet exist (i.e. values do not exist for referenced variables) `:missing-patterns` is returned. These return values are used by various control strategies that can be user-defined.

The current implementation of the knowledge representation language assumes the existence of a rule-group definition that must include the slots: `*lhs-tickled-queue*` and `*rhs-tickled-queue*`. These slots are used to queue up rules whose LHS and RHS (respectively) are potentially `:ok`. This allows a wide variety of control strategies to be defined including forward and backward chaining as well as various combinations thereof.

The KBMS implemented here defines a knowledge base to consist of a number of rule groups as well as domain knowledge. Each rule group provides mechanisms for defining aspects of the inference strategy using either further rule groups or user-defined functions. A rule group also has a mechanism for specifying the level of determinism desired over rule execution ([1, 9]). Basically, the rule group inference strategy consists of a match, evaluate, and fire loop. The match phase is supported by the database interface which automatically queues up potential rules on the tickled-queues. The evaluate phase then checks the rules on the tickled-queues to determine which ones belong in the conflict set. Finally, one rule is selected and fired. This approach is completely non-deterministic in determining which rules get fired from the conflict set. In between the match and evaluate phases a control phase is added. This control phase consists of the definition of a regular expression that defines which rules may be fired next. In this implementation the transition table derived from the regular expression is given instead of the regular expression itself. This is much simpler from the user's perspective. For the user defining a number of rules and wanting to insert some control over them, it is easier to specify a transition table over the rules than it is to define the regular expression that the transition table can be derived from. Furthermore, it is easier to maintain a transition table than a regular expression during rule group modification.

The knowledge representation language, although not completely independent of the KBMS, defines a very general mechanism for specifying knowledge and inference. The KBMS defined here is also very general and allows for a wide variety of specification of inference strategies over the knowledge. This can be done using the knowledge language or by alternatively writing executable code directly (i.e. interpreted vs. compiled).

2.4. The Tractability of the KBMS

Considering the generality and expressivity of the knowledge language and the KBMS, efficiency has the potential of getting lost. There are three ways of maintaining tractability in the knowledge language.

The first way of maintaining tractability is to manage the match phase of the inference cycle efficiently. The conceptual definition of the match phase is to check each rule and evaluate its LHS (RHS) and if it is `:ok` then to queue it up. Obviously, this is also the slowest approach. The step this implementation takes is to compile rules into a rule constraint network that maintains rules in a form more suited for recognizing when data becomes available that has the potential of satisfying the LHS (RHS) of a rule. Consider the LHS of a rule. During evaluation, all the referenced variables must have values in order to determine if the LHS is satisfied. The rule constraint network represents variables as nodes in the network and rules as rule-nodes (see Figure 2). As variables get values, the nodes representing the variables are triggered. All the rule-nodes connected to the node are then triggered. These rule-nodes are then checked to see if all the nodes representing the LHS (RHS) variables have a value. If so, the rule represented by the rule-node is then put on the appropriate queue. As can be seen, this method allows rules to be tickled that may not be satisfied. The rule constraint network only checks to see if variables have values, not if they have the right value. Thus the rules on the tickled queues must still be evaluated for satisfaction.

Quantified rules are also compiled into this form as the variables being quantified over are given values. Thus, if a rule asks if there exists a symptom in the symptom-set and there are three symptoms in symptom-set, then three instances of this graph structure will be created, one of which may cause the rule to fire.

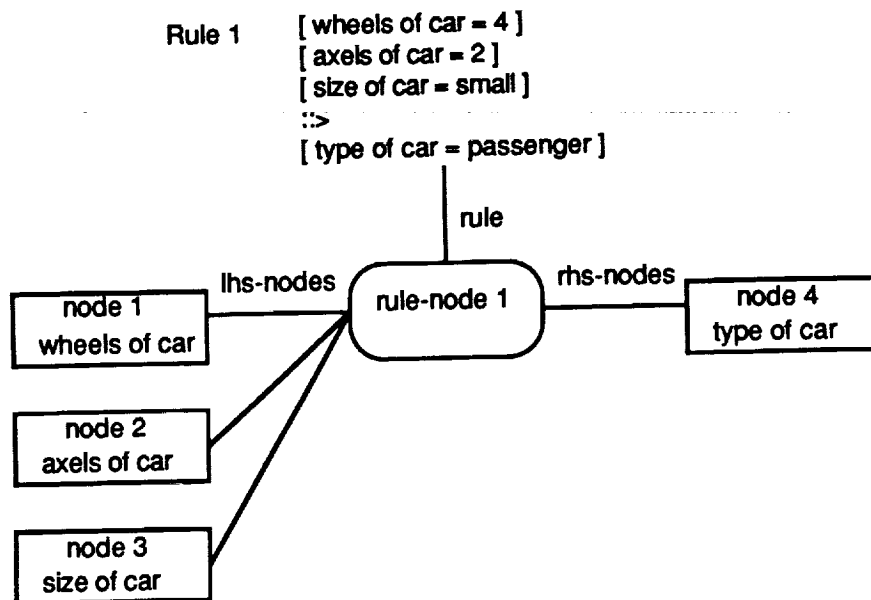


Figure 2 – Rule Constraint Network Example

Alternatively, rules could be compiled more completely so that variables are checked to see if they have the right value. The complexity of the knowledge language implemented here makes this a difficult task and it has not been determined that it would be cost effective. Languages with less expressivity can be completely compiled much more easily (such as OPS5 using the RETE network [7, 8]). The expressivity and tractability trade-off turns up once again.

The second way of maintaining tractability is by providing code for inference strategies instead of providing rules defining the inference strategy. This is making use of the compiled vs. interpreted option. This means that a programmer fluent in the language that the knowledge language is implemented in needs to be available for both implementation and maintenance. However, considering the expressivity of the knowledge language, this may be a cost effective solution. It is very easy for the programmer to express what is desired without having to go into contortions over representational limits.

The third way to maintain tractability is by the effective use of knowledge. This is to make use of Raj Reddy's fifth principle: "Knowledge eliminates the need for search" ([16] see also [12]). In other words, the domain to be represented needs to be analyzed for maximum efficiency in terms of knowledge organization. For example, I can have two rule groups; one rule group forward chains on various data and computes a value for the variable *diagnosis*. The other rule group then uses the value of *diagnosis* to output the results to the user. If there are 50 rules in each rule group that use the variable *diagnosis*, then 100 rules are triggered whenever the value of the variable changes. If both of these rule groups are made to be sub-rule groups of a control rule group, the first rule group can compute a value for *diagnosis*. The control rule group can then set the value for the variable *the-diagnosis* as the value of *diagnosis*. *the-diagnosis* is then used by the second rule group instead of *diagnosis*. Now only 50 rules get triggered at one time.

The representation of knowledge should make maximum use of divide and conquer principles of knowledge organization. Another approach along the same lines is to provide strong heuristic knowledge to eliminate the need for search. This can come in many forms including domain guided inference strategies over rule groups as well as judicious use of control over the execution of rules in a rule group (i.e. the transition table method over rules).

3. A Power Management and Distribution Knowledge Agent

In this example a knowledge agent for managing power and distribution for Space Station Freedom is presented. The knowledge base consists of approximately 150 rules at this time. Naturally, space limitations prohibits the presentation of the entire knowledge base. This example should be sufficient to illustrate the representational capabilities of the knowledge language and how it can be applied to defining multiple agents for distributed processing tasks.

The first part of this example consists of the definitions required by the knowledge language to support the KBMS. The second part then describes the Power Management and Distribution Knowledge Agent.

3.1. Definitions for KBMS Support

These definitions define the user-accessible structure of a knowledge base and of a rule group. The lisp frame is for defining various functions that may be called from rules.

```
(frame :name knowledge-base
      :slots ((rule-groups :value nil)
              (agents :value nil)
              (name :value nil)))

(frame :name rule-group
      :slots ((rules :value nil)
              (name)
              (quantified-vars)
              (rg-var)
              (plan)
              (plan-state)
              (plan-table)
              (viable-set :value nil)
              (not-yet-viable-set :value nil)
              (fire-set :value nil)
              (local-variables)
              (conflict-set :value nil)
              (tickle-set :value nil)
              (satisfied-set :value nil)
              (unsatisfied-set :value nil)
              (cant-fire-set :value nil)
              (fired-set :value nil)
              (untickled-set :value nil)
              (*lhs-tickled-queue* :value nil)
              (*rhs-tickled-queue* :value nil)
              (termination-condition :value nil)
              (control-strategy :value
                                #'default-control-strategy)
              (conflict-resolution-strategy :value
                                #'default-conflict-resolution-strategy)
              (execute :value #'execute)))

(frame :name lisp
      :slots ((evaluate :value #'evaluate)
              (evaluate-lhs :value #'evaluate-lhs)
              (evaluate-rhs :value #'evaluate-rhs)
              (interpret-rhs :value #'interpret-rhs)
              (interpret-lhs :value #'interpret-lhs)
              (interpret-else :value #'interpret-else)
              (first :value #'first)))
```



```
(second :value #'second)
(format :value #'format)
(length :value #'length))
```

3.2. Power Management and Distribution Knowledge Agent

The main knowledge base is defined here. It is called pmad and uses a domain file (domain.lisp) to define the various data structures (domain knowledge) relevant to the knowledge agent. The knowledge base consists of three rule groups; a control rule group for controlling the diagnosis of hard faults, a hard fault rule group for computing a diagnosis, and a diagnosis rule group simply for printing a diagnosis.

KB : pmad

DOMAIN : domain.lisp

RULE-GROUP : control-rg

This control rule group currently manages the collection of fault information for diagnosis. It controls the execution of two rule groups: hard-fault and diagnosis, for performing and printing out diagnosis information respectively. Eventually rules will be added to this knowledge agent for soft fault and incipient fault analysis.

```
CONTROL : ((start (Control-Rule1))
            (Control-Rule1 (Control-Rule2))
            (Control-Rule2 (Control-Rule3 Control-Rule4
                                   Control-Rule5))
            (Control-Rule3 (Control-Rule3 Control-Rule4
                                   Control-Rule5))
            (Control-Rule4 (Control-Rule3 Control-Rule4
                                   Control-Rule5))
            (Control-Rule5 (Control-Rule6))
            (Control-Rule6 (Control-Rule1)))
```

```
Control-Rule1
THERE EXISTS symptom-set in symptom-set-queue
< ::>
  [ the-symptom-set = symptom-set ]
  [ diagnosis-set = empty ] >
;
Control-Rule2
::>
[ clusters = power-domain :: cluster-symptoms
  ( the-symptom-set ) ]
;
Control-Rule3
THERE EXISTS cluster in clusters
< ::>
  [ symptoms of symptom-set1 = symptoms of cluster ]
  [ symptom-set1 = cluster ] >
ELSE
[ ready-to-diagnose = false ]
;
Control-Rule4
THERE EXISTS symptom in symptoms of symptom-set1
< [ hard-fault :: execute ( hard-fault ) = :ok ]
::>
[ diagnosis-set = diagnosis-set PLUS diagnosis ]
```

```

    [ the-diagnosis = diagnosis ]
    [ diagnosis = :unknown ]
    [ diagnosis-rg :: execute ( diagnosis-rg ) ]
    [ clusters = clusters MINUS symptom-set1 ] >
;
Control-Rule5
[ ready-to-diagnose = false ]
[ FOR ALL diagnosis1 in diagnosis-set
  < [ diagnosis1 = diagnosis-2 ]
    OR
    [ diagnosis1 = diagnosis-31 ] > ]
::>
[ the-diagnosis = diagnosis-no-power ]
[ diagnosis-set = diagnosis-set PLUS the-diagnosis ]
[ ready-to-diagnose = true ]
ELSE
[ ready-to-diagnose = true ]
;
Control-Rule6
[ ready-to-diagnose = true ]
::>
[ diagnosis-rg :: execute ( diagnosis-rg ) ]
[ symptom-set-queue = symptom-set-queue
  MINUS the-symptom-set ]
[ symptom-set1 = :unknown ]
[ clusters = :unknown ]
[ ready-to-diagnose = :unknown ]
;

```

RULE-GROUP : hard-fault.rg

This rule group takes information about a fault and, after determining if any testing needs to be done and doing it, determines the fault and assigns diagnosis a value.

As can be seen, the control that is specifiable over a rule group may be quite complex. The whole transition table for this rule group is not given (although this is one-third of it, for a rule group consisting of about 90 rules), but the expressivity is still quite apparent.

```

CONTROL : ((start (init-rule))
  (init-rule (Rule1))
  (Rule1 (Rule2))
  (Rule2 (Rule3 Rule4.1 Rule5 Rule31 Rule31.1
    Rule31.2 Rule35 Rule35.1))
  (Rule3 (Rule4))
  (Rule4 (Rule20 Rule32))
  (Rule32 (Rule33))
  (Rule33 (Rule34))
  (Rule4.1 (Rule4.2))
  (Rule4.2 (Rule4.3))
  (Rule4.3 (Rule4.4 Rule4.5 Rule4.6 Rule4.7
    Rule4.8))
  ...
)

Init-rule
::>
[ symptom-set = symptoms of symptom-set1 ]
[ possible-top-switches = empty ]

```

```

[ tripped-top-switches = empty ]
;
Rule1
::>
[ top-symptoms = power-domain ::
    top-symptoms ( symptom-set ) ]
;
Rule2
[ lisp :: length ( top-symptoms ) > 1 ]
::>
[ type = multiple-tops ]
ELSE
[ type = single-top ]
;
Rule3
[ type = multiple-tops ]
[ THERE EXISTS symptom in top-symptoms
    < [ lisp :: length
        ( switches-below of switch of symptom ) > 0 ] > ]
[ FOR ALL symptom in top-symptoms
    < [ fault of symptom = fast-trip ]
    OR
    [ fault of symptom = over-current ] > ]
::>
[ FOR ALL symptom in top-symptoms
    < [ tripped-top-switches = tripped-top-switches PLUS
        switch of symptom ] > ]
[ THERE EXISTS symptom in top-symptoms
    < [ possible-top-switches = switch of symptom PLUS
        siblings of switch of symptom ] > ]
[ THERE EXISTS symptom in top-symptoms
    < [ trip-type = fault of symptom ] > ]
[ type = multiple-top-current-trip ]
;
...
Rule4.4
[ type = multiple-tops ]
[ lisp :: length ( new-symptoms ) = 1 ]
[ THERE EXISTS symptom in new-symptoms
    < [ THERE EXISTS symptom1 in top-symptoms
        < [ switch of symptom = switch of symptom1 ]
        [ fault of symptom = fault of symptom1 ] > ] > ]
::>
[ diagnosis = diagnosis-54 ]
;
...
:
[ diagnosis ]

```

RULE-GROUP : diagnosis.rg

This rule group simply prints out some statements and communicates diagnostic information to the scheduling knowledge agent depending on the particular diagnosis encountered.

```

d-rule1 @@ backrush in load center
[ the-diagnosis = diagnosis-54 ]
::>
[ the-diagnosis = :unknown ]

```

```

[ lisp :: format ( t "~%The following load center
                    RPCs tripped on fast-trip~%" ) ]
...
;
...
:
[ the-diagnosis = :unknown ]

```

The rest of the knowledge base for this knowledge agent.

```

Domain-Knowledge :
constants :
  t ; :ok ; :ng ; :missing-patterns ; yes ; no ;
  hard-fault ;
  diagnosis-rg ;
  multiple-tops ;
  single-top ;
  multiple-top-current-trip ;
  over-current ;
  under-voltage ;
  fast-trip ;
  ground-fault ;
  diagnosis-1 ;
  diagnosis-2 ;
  ...
  diagnosis-no-power .
facts :
  empty = ( ) .
frames :
  (fcreate-instance 'symptom-set 'symptom-set1) .

```

Begin : control-rg

END-KB

```

+++++
domain.lisp
+++++

```

```

(frame :name power-domain
      :slots ((top-symptoms :value #'top-symptoms)
              ...
              (close-switch :value #'close-switch)))

(frame :name symptom-set
      :slots ((symptoms)))

(frame :name symptom
      :slots ((switch) (fault)))

(frame :name switch
      :slots ((name)
              (type)
              (current)
              (switches-below :value nil)
              (switch-above :value nil)
              (siblings)

```

```
...  
(current-rating)  
(fast-trip-percent)))
```

...
Lots of domain knowledge here to build instances of switches and sensors, etc. Knowledge of the topology is encoded here. About 30k worth.

4. Conclusions and Future Work

An architecture for defining and modifying knowledge base management systems that may be used for applications in distributed AI has been presented. The architecture is very flexible and relatively efficient. It has been used to define three very different knowledge agents: One for solving a toy problem to compute how to send a package consisting of about ten rules; One for solving the monkeys and bananas problem consisting of about twenty fairly complex rules, this one was directly adapted from a solution given for OPS5; Finally the agent given in this paper and consisting of around 150 rules.

The results we have noticed so far have shown that this architecture provided the ability to easily implement a solution to a wide variety of problems. The monkeys and bananas problem has driven out many areas of weakness in the implementation that are being dealt with. The speed with which this architecture solves the monkeys and bananas problem is hardly even comparable to that of OPS5 at this point. However, the result of implementing our fault diagnosis problem for power management and distribution has turned out very well. Using simple forward chaining and lots of control knowledge in the hard fault rule group has enabled us to provide a solution that is very fast and easily maintainable. The maintainability is very important for this domain as the requirements for Space Station Freedom have not been completely specified.

4.1. Future Work

This system is being implemented as part of a much larger system, KNOMAD (Knowledge Management and Design System). We have identified a number of areas where knowledge needs to be added to support a completely robust, domain independent environment for specifying knowledge based systems. These include the addition of a constraint system, a temporal database, and analytical and qualitative reasoning. These additions will then support planning, scheduling, and causal reasoning at the least. Adding these components must involve how the KBMS will access and use these components as well as how these components will use the existing database and database interface mechanisms.

Work also needs to be pursued to determine the possibility of adding RETE-like structures as a part of the rule constraint network.

5. Acknowledgements

This work is being supported by NASA, Marshall Space Flight Center, contract NAS8-36433.

6. References

- [1] Baskin, A.B., "Combining Deterministic and Non-deterministic Rule Scheduling in an Expert System," AAMSI, 1986.
- [2] Bond, Alan H., and Les Gasser, Eds., "Readings in Distributed Artificial Intelligence," Morgan Kaufman, 1988.

- [3] Brodie, M.L., J. Mylopoulos, and J.W. Schmidt, (Eds.), "On Conceptual Modelling," Springer-Verlag, New York, 1984.
- [4] Buchanan, Bruce G., and Richard O. Duda, "Principles of Rule-Based Expert systems," Stanford Heuristic Programming Project Report No. HPP-82-14, 1982.
- [5] Carriero, Nicholas, and David Gelernter, "Linda in Context," Communications of the ACM, 32(4), 1989.
- [6] D'Angelo, Antonio, Giovanni Guida, Maurixo Pighin, and Carlo Tasso, "A Mechanism for Representing and Using Meta-Knowledge in Rule-Based Systems," Approximate Reasoning in Expert Systems, 1985.
- [7] Forgy, Charles L., "RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," AI 19(1) 1982.
- [8] Forgy, Charles L., and Susan J. Shepard, "RETE: A Fast Match Algorithm," AI Expert, Jan. 1987.
- [9] Georgeff, M.P., "Procedural Control in Production Systems," AI 18, pp. 175-201, 1982.
- [10] Hayes, P.J., "The Logic of Frames," in Webber, B.L., and Nils J. Nilsson (Eds.) *Readings in Artificial Intelligence*, pp. 451-458, 1981.
- [11] Hayes-Roth, Frederick, "Towards Benchmarks for Knowledge Systems and Their Implications for Data Engineering," IEEE Transactions on Knowledge And Data Engineering, Vol. 1, No. 1, March 1989.
- [12] Lenat, Douglas B., and Edward A. Feigenbaum, "On the Thresholds of Knowledge," International Workshop on Artificial Intelligence for Industrial Applications, 1988.
- [13] Levesque, Hector J., "Knowledge Representation and Reasoning," Annual Reviews of Computer Science, 1986.
- [14] Matsuoka, Satoshi, and Satoru Kawai, "Using Tuple Space Communication in Distributed Object-Oriented Languages," OOPSLA 1988 Proceedings.
- [15] Minsky, M., "A Framework for Representing Knowledge," in P. Winston (Ed.) *The Psychology of Computer Vision*, McGraw-Hill, New York, pp. 211-277, 1975.
- [16] Reddy, Raj, Presidential Address at the American Association for Artificial Intelligence Conference, 1986.
- [17] Reiter, R., "Towards a Logical Reconstruction of Relational Database Theory," in [3], pp. 191-233, 1984.
- [18] Stefik, Mark, Jan Aikins, Robert Balzer, John Benoit, Lawrence Birnbaum, Frederick Hayes-Roth, and Earl Sacerdoti, "The Organization of Expert Systems, A Tutorial," AI 18, 1982.
- [19] Wilkins, David E., "Practical Planning: Extending the Classical AI Planning Paradigm," Morgan Kaufman, 1988.